Microsoft Kinect

Criando aplicações interativas com o Microsoft Kinect

Casa do Código

GABRIEL SCHADE CARDOSO

Sumário

Capítulo 1

Introdução à Interfaces Naturais

Formas inovadoras de interação com o usuário têm sido propostas por interfaces focadas ao paradigma de interface natural ao longo dos anos. Os softwares em diversas áreas não devem exigir que o usuário tenha grande conhecimento em computação para usá-lo, devem sempre buscar por simplicidade e facilidade.

A primeira interface com o usuário a aparecer foi a CLI (Command Line Interface) que utiliza comandos de entrada textuais específicas para desempenhar funções em um aplicativo, algo que exigia uma grande curva de aprendizado. Algum tempo depois surgiu a famosa GUI (Graphical User Interface) que utiliza janelas gráficas, mouse, botões e diversos outros componentes visuais, arrisco dizer que graças à GUI os computadores conseguiram atingir tantas pessoas e se popularizar tão rápido. Hoje em dia este conceito é o mais utilizado em sistemas operacionais e aplicações.

Devido à quantidade de usuários e demanda por aplicações cada vez mais intuitivas e simples surgiu o conceito de NUI (Natural User Interface) ou simplesmente interfaces naturais. Este tipo de interface foca em utilizar uma linguagem natural para a interação humana com o aplicativo, como gestos, poses e comandos de voz.

1.1 Apresentando o Kinect

Como proposta de interface natural inicialmente focada para a área de jogos, a empresa Microsoft em parceria com a empresa israelita PrimeSense construiu o sensor de movimentos que hoje é chamado de Kinect. O hardware do Kinect teve boa parte herdado pelo hardware do PrimeSense. Na figura 1.1 podemos visualizar o "irmão mais velho" do Kinect.

Figura 1.1: Sensor da PrimeSense

O Kinect certamente causou uma revolução na área de interações com jogos, pois a partir de agora, não é mais necessário utilizar um controle: o sensor capta movimentos e comandos de voz do usuário, abrindo um leque de possibilidades totalmente novo, utilizando a linguagem natural para a interação com os jogos.

Inicialmente o sensor Kinect era conhecido pelo codinome Projeto Natal, fazendo referência à cidade brasileira Natal, isso ocorreu devido ao fato de que um dos idealizadores do projeto foi um brasileiro chamado Alex Kipman.

A figura 1.2 exibe o sensor concebido pela Microsoft, podemos notar que, apesar de um design mais elegante e robusto ele possui grandes semelhanças com o PrimeSense.

Figura 1.2: Microsoft Kinect

Em fevereiro de 2011, quatro meses após o sensor ter sido lançado no Brasil, a Microsoft anunciava o lançamento oficial de um SDK (software development kit) que pode ser obtido de forma gratuita. Com este kit de desenvolvimento a Microsoft permite que desenvolvedores possam criar aplicativos para computadores nas linguagens C++, C# e Visual Basic utilizando o hardware Kinect, ou seja, a limitação de que o Kinect era um dispositivo apenas para a área de jogos não era mais verdadeira.

Atualmente existem três tipos diferentes de Kinect, iremos diferenciá-los pelos seguintes nomes: **Kinect**, **Kinect for Windows** e **Kinect for Xbox 360**. O sensor conhecido comercialmente como Kinect for Windows possui um hardware diferenciado, seus microfones possuem uma melhor qualidade e ele é capaz de rastrear o usuário mais próximo ao sensor ou quando ele estiver sentado. Explicaremos melhor sobre isso ao decorrer do livro.

A diferença mais importante que você deve ter em mente agora é entre o Kinect for Xbox 360 e os outros. Esta diferença trata-se do cabo de conexão do sensor, esta versão é vendida somente junto com um console Xbox 360 e seu cabo de conexão possui uma entrada específica, caso seu sensor seja deste tipo é necessário comprar um adaptador para conectar o Kinect em um computador, a figura 1.3 ilustra o cabo e o adaptador para este tipo de sensor.

Figura 1.3: Kinect for Xbox 360 e seu adaptador

O sensor possui um hardware que oferece diversos recursos que iremos explorar ao longo do livro, os principais são: Emissor de luz infravermelho, sensor RGB, sensor infravermelho, eixo motorizado e um conjunto de microfones dispostos ao longo do sensor. A figura 1.4 apresenta uma visão do interior do sensor com indicativos de onde estão localizados estes recursos.
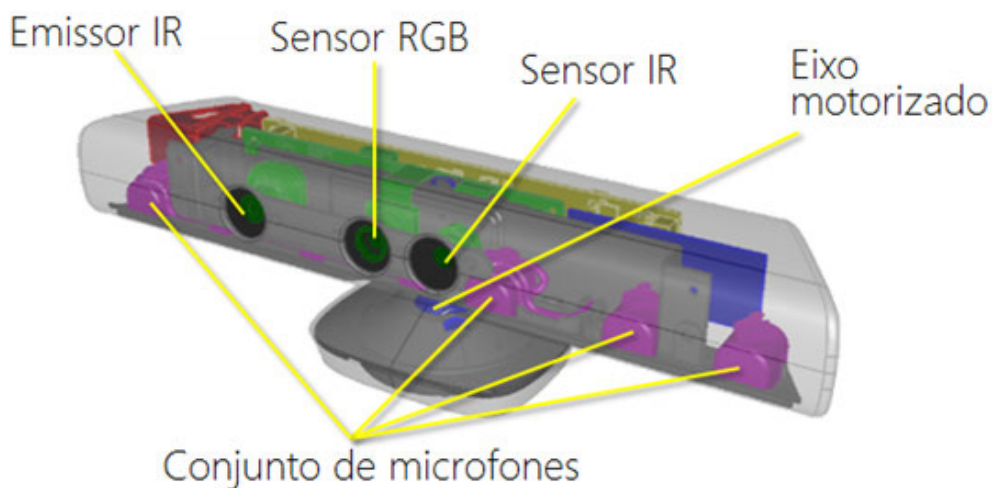
Figura 1.4: Kinect por dentro

1.2 O que você encontrará neste livro

Este livro é escrito para desenvolvedores que já possuem conhecimento referente aos principais conceitos relacionados à orientação a objetos e que já tiveram contato com à linguagem C#, para facilitar a compreensão de desenvolvedores menos experiêntes nessa linguagem será dado pequenas explicações sobre alguns conceitos da linguagem. Nos capítulos iniciais iremos desenvolver aplicações de testes e a partir do capítulo 4 iremos construir uma aplicação até o fim do livro. Estas implementações serão focadas em conceitos de interfaces naturais específicos utilizando um sensor Kinect. As aplicações criadas neste livro não irão levar em consideração padrões arquiteturais e melhorias de performance, será levado em consideração a clareza e facilidade de entendimento do leitor e todas as implementações terão seu código centralizado no repositório: https://github.com/gabrielschade/CrieAplicacoesInterativascomoMicrosoftKinect.

Ao longo deste livro nos capítulos mais avançados iremos construir uma aplicação utilizando todos os conceitos e recursos estudados, no fim do livro teremos uma aplicação completa utilizando o Kinect.

A grande meta deste livro é encorajar e iniciar o leitor a ingressar neste mundo desafiador (e muito divertido!) da programação seguindo os conceitos de NUI através do sensor Kinect, o que invariavelmente irá envolver bastante teoria e prática, de

uma forma fluída, simples e divertida!

1.3 Antes de começar

Antes de começarmos a falar da nossa primeira aplicação, é necessário que você tenha feito o download do SDK, ele é gratuito e pode ser baixado através deste link: http://www.microsoft.com/en-us/kinectforwindows/develop/developer-downloads.aspx. Além da ultima versão do SDK, também é aconselhável já fazer o download do *developer toolkit* e do *Microsoft Speech Platform SDK* para utilizarmos em nossas futuras implementações.

Neste livro utilizaremos o **Visual Studio 2012** como IDE de desenvolvimento, caso você ainda não possua esta ferramenta, existe uma versão gratuita que pode ser baixada através do link: http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-products.

É importante lembrar que caso você possua um Kinect for Xbox 360 é preciso que você tenha um adaptador para conectá-lo ao seu computador.

Capítulo 2

Primeira aplicação com o sensor

Neste capítulo aprenderemos a codificar e testar uma aplicação que utilize o sensor Kinect. Esta aplicação será bastante simples e terá o intuito de termos o primeiro contato com este tipo de programação. Ela terá como função reconhecer se o usuário está com a mão direita acima da altura de sua cabeça e notificar o usuário exibindo uma mensagem em um diálogo quando isso acontecer, simples não?

A primeira coisa que temos que fazer é criar nosso projeto do tipo *WPF Application* no Visual Studio e adicionar a referência para o SDK do Kinect.

WPF Applications

Windows Presentation Foundation ou simplesmente WPF é um tipo de projeto proposto no Visual Studio para aplicações desktop para Windows, apesar do Kinect não se limitar a aplicações WPF, existem diversas ferramentas inclusas no SDK que nos auxiliam neste tipo de projeto. Para saber mais sobre WPF acesse: http://msdn.microsoft.com/pt-br//library/ms754130.aspx

Para adicionar a referência para o SDK do Kinect basta pressionar o botão direito do mouse sobre o seu projeto e selecionar a opção "*Add Reference...*". Será aberta uma janela para selecionar o assembly que deseja adicionar em seu projeto, pesquise pela palavra Kinect, selecione o assembly Microsoft.Kinect e pressione o botão OK, conforme ilustra a figura 2.1. Com isso nosso projeto já está preparado para a implementação, vamos codificar!
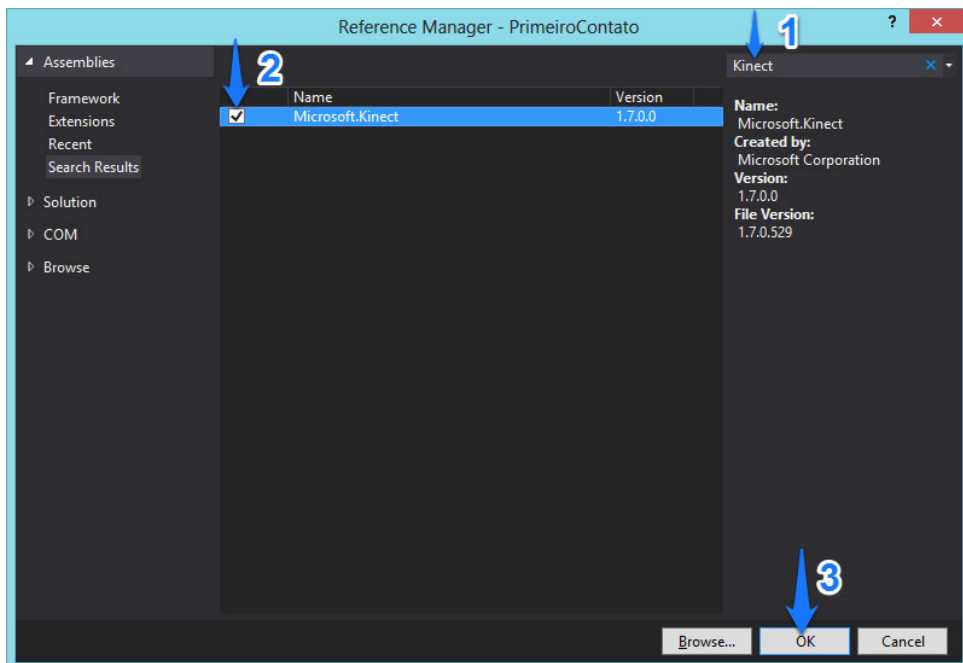
Figura 2.1: Adicionando referência ao assembly Microsoft.Kinect

2.1 Criando o projeto AuxiliarKinect

Antes da implementação de qualquer regra é necessário implementar o processo de configuração do sensor Kinect que está conectado ao computador, não iremos tratar agora cenários de exceção, então para que uma situação inesperada não ocorra em sua aplicação, tenha certeza que seu Kinect está devidamente conectado ao seu computador e à energia.

Para facilitar a configuração do sensor para as próximas aplicações iremos criar um projeto separado para nos auxiliar em algumas tarefas comuns do Kinect. Para fazer isso abra uma nova instância do Visual Studio e crie um projeto do tipo *Class Library*. Este projeto irá gerar nossa DLL que conterá diversas funções do Kinect, vamos chamá-lo de AuxiliarKinect.

Como descrevemos anteriormente neste projeto também é necessário a referência para o SDK do Kinect. Note que no projeto já existe uma classe chamada *Class1*, iremos excluí-la. Pressione o botão direito sobre o projeto e crie uma nova pasta chamada *FuncoesBasicas*.

Antes de começarmos a implementar é importante ter em mente que não existe limitação no que diz respeito à quantidade de sensores conectados em seu computador simultaneamente. É possível identificar os diferentes sensores por um identificador único que cada sensor possui, mas neste livro iremos focar no cenário mais comum, onde as aplicações utilizam apenas um sensor.

Pensando neste cenário, a Microsoft criou na classe **KinectSensor** a propriedade chamada **KinectSensors**. Esta propriedade é uma coleção de objetos do tipo `KinectSensor`, ou seja, todos os sensores conectados no computador serão listados nesta coleção. Tendo conhecimento desta propriedade e sabendo que possa existir mais de um sensor, é fácil perceber que é necessário fazer uma busca para identificar quantos sensores estão conectados no computador. Este código pode ser escrito de maneira muito simples através de uma expressão lambda.

Expressões Lambdas

Expressão lambda é um recurso muito poderoso disponível na linguagem C#, trata-se de uma notação para escrita de funções anônimas que podem realizar as mais diversas tarefas, como suporte para a linguagem de consulta LINQ e escrita simplificada de métodos recursivos, por exemplo. Para mais informações sobre expressões lambda acesse o link: http://msdn.microsoft.com/pt-br/library/vstudio/bb397687.aspx

Sabendo que utilizaremos apenas um sensor na aplicação podemos fazer uma busca na coleção de sensores para obtermos o primeiro Kinect conectado ao computador (que neste caso sempre será o único). Após encontrar o sensor é preciso inicializá-lo através do método `Start` da classe `KinectSensor`. Feito a inicialização iremos forçar o Kinect a manter o eixo motorizado do sensor de acordo com uma informação parametrizada. É importante saber que, para alterar o ângulo do eixo motorizado do sensor, é necessário inicializá-lo antes, caso contrário uma exceção será disparada.

Agora criaremos a classe `InicializadorKinect` dentro da pasta *FuncoesBasicas*. Além de ter adicionado a referência ao assembly *Microsoft.Kinect* adicione também a diretiva *using* para este mesmo assembly dentro da classe `InicializadorKinect`. Se tudo estiver correto você já poderá acessar a classe `KinectSensor`, esta classe é de extrema importância, pois ela é a representação virtual do sensor Kinect em sua aplicação.

A classe `InicializadorKinect` deve ser pública para podermos acessá-la em nossa aplicação, portanto não esqueça de utilizar a palavra reservada `public` na declaração da classe. Vamos criar um método estático para fazer as operações descritas anteriormente (inicializar o sensor e configurar seu ângulo de elevação). Nosso método se chamará `InicializarPrimeiroSensor` ele irá receber por parâmetro o ângulo de elevação inicial do sensor e deverá retornar o primeiro sensor conectado no computador.

```
public static KinectSensor
            InicializarPrimeiroSensor(int anguloElevacaoInicial)
{
    KinectSensor kinect =
    KinectSensor.KinectSensors.First(sensor =>
```

```
    sensor.Status == KinectStatus.Connected );
    kinect.Start();
    kinect.ElevationAngle = anguloElevacaoInicial;

    return kinect;
}
```

Notou como foi simples? Apenas com algumas linhas de código podemos obter o nosso sensor, iniciar o seu processamento e alterar a angulação do eixo motorizado. Agora você deve colocar uma chamada para este método em nossa aplicação.

Procure pela classe MainWindow, apesar de ela ser uma janela não iremos alterar em nada a parte visual, então se preocupe apenas com o arquivo C# (MainWindow.xaml.cs). Além de ter adicionado a referência ao assembly *Microsoft.Kinect* adicione também a referência para a DLL que criamos no projeto `AuxiliarKinect`. Lembre-se também de utilizar a diretiva *using* para estes mesmos assemblys dentro da classe MainWindow.

Nesta aplicação, além de inicializarmos com a configuração padrão do Kinect precisamos habilitar o rastreamento de esqueleto, pois é isto que irá nos permitir verificar a posição espacial da mão e da cabeça do usuário. Então vamos criar um método chamado `InicializarKinect` para fazer as operações supracitadas.

```
private void InicializarSensor()
{
    KinectSensor kinect = InicializadorKinect
                          .InicializarPrimeiroSensor(10);

    kinect.SkeletonStream.Enable();
}
```

Com nossa DLL, ficou muito simples inicializar o sensor! Agora no construtor da classe após a chamada do método `InitializeComponent` que já está presente por padrão em todas as janelas, crie uma chamada para este método. Depois conecte o sensor ao seu computador e execute a aplicação, viu o que aconteceu? A aplicação já está reconhecendo o sensor e posicionando o ângulo de elevação de seu eixo motorizado!

2.2 Criando a regra de nossa aplicação

Para armazenar o resultado da comparação de altura entre a mão direita do usuário e sua cabeça iremos criar a propriedade `bool MaoDireitaAcimaCabeca`. Já con-

figuramos o sensor e criamos a propriedade que irá armazenar o resultado de nossa regra, mas ainda não escrevemos o código que, de fato, valida esta regra e exibe a mensagem para o usuário. Onde esta regra deve ser escrita? Antes de respondermos precisamos entender mais um conceito envolvido na programação com o Kinect.

Não iremos entrar em muitos detalhes nesta primeira aplicação, mas é importante saber que o Kinect atualiza um conjunto de informações em forma de quadros ou *frames*. Em sua configuração padrão, a cada segundo o Kinect cria 30 quadros, e cada quadro possui seu conjunto próprio de informações.

Agora criaremos o método que executa nossa regra, para fazer esta validação precisamos das informações referentes ao esqueleto do usuário. Elas estão encapsuladas na classe `SkeletonFrame`, então o método que irá executar a regra deve receber um objeto deste tipo por parâmetro.

O rastreamento de esqueleto do usuário do Kinect pode identificar até seis esqueletos simultâneos e todos estes esqueletos estão no objeto da classe `SkeletonFrame`. Teremos que extrair a informação sobre os esqueletos do quadro e logo em seguida, vamos buscar pelo primeiro esqueleto de usuário que o Kinect esteja visualizando — caso exista algum, faremos a validação para nossa regra e exibiremos a mensagem. O código a seguir descreve como este método deve ser escrito.

```
private void ExecutarRegraMaoDireitaAcimaDaCabeca
 (SkeletonFrame quadroAtual)
{
    Skeleton[] esqueletos = new Skeleton[6];
    quadroAtual.CopySkeletonDataTo(esqueletos);

    Skeleton usuario =
    esqueletos.FirstOrDefault( esqueleto =>
    esqueleto.TrackingState == SkeletonTrackingState.Tracked );

    if (usuario != null)
    {
        Joint maoDireita = usuario.Joints[JointType.HandRight];
        Joint cabeca = usuario.Joints[JointType.Head];
        bool novoTesteMaoDireitaAcimaCabeca =
            maoDireita.Position.Y > cabeca.Position.Y;

        if ( MaoDireitaAcimaCabeca !=
             novoTesteMaoDireitaAcimaCabeca)
```

```
        {
            MaoDireitaAcimaCabeca =
             novoTesteMaoDireitaAcimaCabeca;

            if (MaoDireitaAcimaCabeca)
                MessageBox
                  .Show("A mão direita está acima da cabeça!");
        }
    }
}
```

No código anterior copiamos todas as informações dos esqueletos de usuário para um array, depois verificamos se existe algum esqueleto que possua o estado *tracked* (rastreado), caso exista executamos nossa regra para este esqueleto.

Agora precisamos inserir uma chamada a este método em algum lugar da nossa aplicação, onde devemos fazer isso?

Cada vez que o Kinect processa as informações a respeito do esqueleto de todos os usuários e gera um novo quadro, a classe `KinectSensor` dispara um evento que teremos que interpretar. Neste evento é fornecido um objeto do tipo `SkeletonFrameReadyEventArgs`. Este objeto possui diversas propriedades e entre eles está o objeto `SkeletonFrame` que devemos enviar para nosso método criado anteriormente. Portanto, no método que interpreta este evento é que devemos fazer uma chamada para o método `ExecutarRegraMaoDireitaAcimaDaCabeca`!

A primeira alteração em nosso código é inserir a linha de código `kinect.SkeletonFrameReady += KinectEvent;` no método `InicializarSensor()` entre a inicialização do rastreamento de esqueleto e a inicialização do sensor. Você pode utilizar o Visual Studio para autogerar o método que interpreta o evento. A única responsabilidade deste método é abrir o novo quadro que o Kinect está enviando e chamar nosso método de regra, conforme código a seguir.

```
private void KinectEvent(object sender, SkeletonFrameReadyEventArgs e)
{
    using (SkeletonFrame quadroAtual = e.OpenSkeletonFrame())
    {
        if (quadroAtual != null)
        {
            ExecutarRegraMaoDireitaAcimaDaCabeca(quadroAtual);
```

```
        }
    }
}
```

Você pode notar que no código anterior verificamos se o quadro atual está nulo. Isso ocorre quando não há informações sobre o novo quadro, ou seja, não há nenhum usuário em frente ao sensor.

Execute sua aplicação e veja o resultado! Agora oficialmente temos a primeira aplicação utilizando o sensor Kinect e seu rastreamento de esqueleto, não é legal?
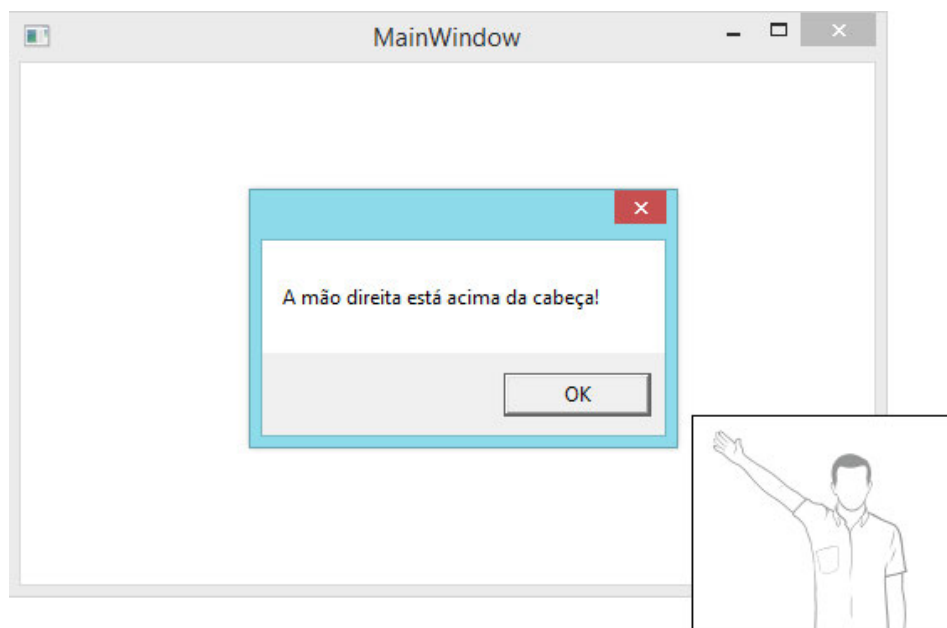
Figura 2.2: Aplicação sendo executada

2.3 Testando e depurando nossa aplicação

Você não terá muitas dificuldades em testar esta aplicação, basta posicionar-se em frente ao sensor e levantar o braço até que sua mão direita esteja acima de sua cabeça, mas sem utilizar a ferramenta certa você pode ter um problema com **depuração**. Como fazer a depuração da nossa aplicação de forma simples, sem precisar nos levantarmos da cadeira e fazer todos os gestos que gostaríamos de testar a cada novo

teste?

A resposta para este problema foi instalada em seu computador junto com o SDK indicado no começo deste livro, ela chama-se **Kinect Studio**. A aplicação Kinect Studio nos permite gravar ações para executar em um teste, sendo assim, podemos gravar o movimento apenas uma vez e testá-lo quantas vezes for necessário.

Após executar sua aplicação execute o Kinect Studio e note que será exibida uma janela. Esta janela deve conter sua aplicação listada, caso sua aplicação não esteja sendo apresentada, certifique-se de que ela realmente está em execução e pressione o botão *Refresh*, após ter feito isso a janela mostrada deve se parecer com a figura 2.3.
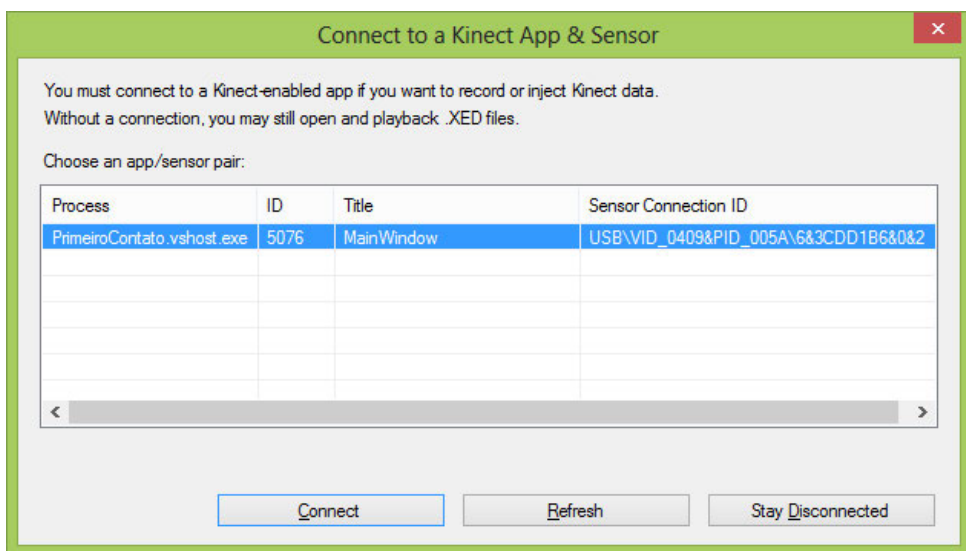
Figura 2.3: Conectando sua aplicação no Kinect Studio

Pressione o botão *Connect* e as quatro janelas que estavam ao fundo são mostradas, na figura 2.4 as três janelas inferiores representam as visões do sensor. A janela *Color Viewer* não está apresentando nenhuma imagem porque não ativamos a câmera de cores do Kinect em nossa aplicação.

Na janela superior estão os controles com os quais você pode gravar, abrir e executar arquivos com a extensão **.xed**. Através desta ferramenta você pode dar o *playback* de seus testes gravados facilitando (e muito) a depuração e o teste de suas aplicações com o Kinect, não irei mais citar esta ferramenta, mas aconselho fortemente utilizá-la ao decorrer do livro.
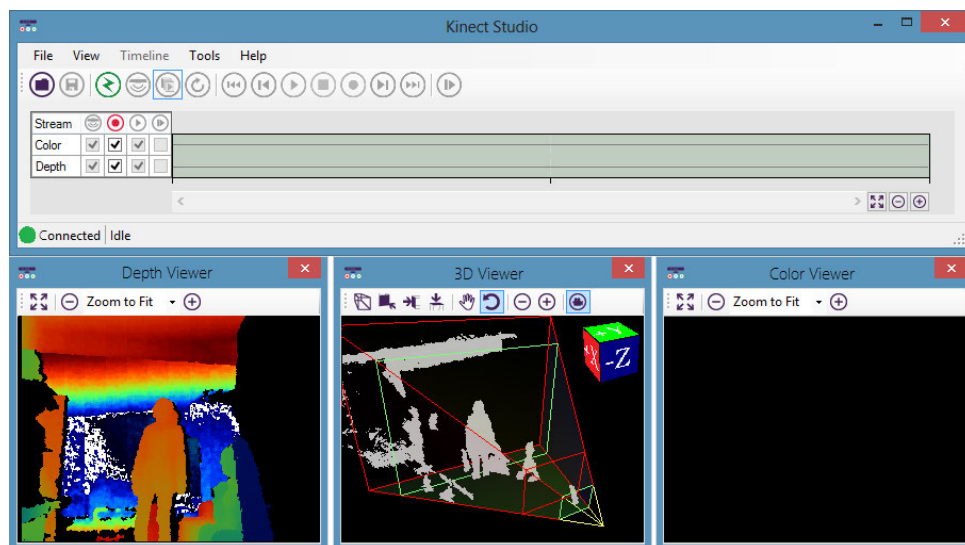
Figura 2.4: Kinect Studio

2.4 Continuando

Com isso você já tem uma noção básica sobre o que é o sensor Kinect, qual seu hardware e como desenvolver e testar uma aplicação que utilize seus recursos. A partir desta etapa o livro irá separar as funcionalidades por capítulos intercalando entre a explicação do hardware e do software envolvido em cada uma destas funcionalidades e ao final de tudo iremos construir uma aplicação que agrega diversos conceitos que iremos ver.

Capítulo 3

Acelerômetro e Eixo Motorizado

Neste capítulo iremos entender como funciona o acelerômetro do Kinect e quando podemos nos beneficiar com este recurso. Além disso, também será explicado sobre o campo de visão do sensor e sobre como podemos adaptar o seu foco utilizando o recurso disponível no SDK para manipulação do eixo motorizado — o que torna este componente, apesar de relativamente simples, muito importante para quem está pensando em uma aplicação NUI com o Kinect.

3.1 Acelerômetro

Um acelerômetro nada mais é que, como o nome já sugere, um sensor para medir a aceleração sobre um objeto. O Kinect possui um acelerômetro capaz de fazer medidas nas três dimensões: X, Y e Z. Além disso ele está configurado para suportar um intervalo de 2g, onde g é a força que a gravidade implica sobre o sensor, isso permite que este pequeno sensor consiga informar a sua orientação em relação à gravidade.

Este sensor de aceleração possui uma sensibilidade para temperaturas, o valor de seus eixos pode variar em até três graus positiva ou negativamente. Para compensar

este desvio é necessário fazer uma comparação entre o eixo Y do acelerômetro e os dados de profundidade.

Ok, mas em que um acelerômetro pode nos ajudar? Bom, esta pergunta pode vir a cabeça neste momento, mas é importante termos em mente que as aplicações do Kinect vão além de um computador ou notebook, existem diversos projetos em que o Kinect é utilizado em um totem ou em um dispositivo que se move, ou seja, o uso do acelerômetro se estende por todos os casos em que orientação do sensor é alterada.

Entendendo o acelerômetro no SDK

As informações do acelerômetro no SDK são disponibilizadas através de um método da já conhecida classe `KinectSensor`. O método `AccelerometerGetCurrentReading()` retorna as informações sobre o acelerômetro em um vetor de 4 posições (X, Y, Z e W) sendo que W sempre terá o valor zero.

Apesar de parecer estranho em um primeiro momento que haja a posição W no retorno de algo que deveria estar em um sistemas de coordenadas de três dimensões, faz total sentido se conhecermos o conceito de coordenadas homogêneas. Trata-se de uma ideia bem simples e poderosa: basicamente você representa as coordenadas X, Y e Z em uma quádrupla ordenada*[**X, Y, Z e W**] onde X = X/W + 1, Y = Y/W + 1 e Z = Z/W +1. Este sistema permite representar projeções conhecidas como pontos impróprios, que são utilizados para representar o feixe de retas paralelas a um outro eixo. Existem diversas outras vantagens para isso ser utilizado, mas não cabe ao escopo deste livro descrever todas.

O sistema de coordenadas do acelerômetro do sensor está centralizado com o dispositivo. Por padrão o retorno deste método, caso o Kinect esteja em uma superfície plana, será próximo a (x:0, y:-1.0, z:0 e w:0).

Infelizmente não há um evento nativo para quando estas informações são alteradas, é necessário monitorarmos manualmente, mas podemos e iremos criar um sistema de eventos para fazer isso.

Aplicação

Vamos criar uma aplicação para medir o resultado do acelerômetro de tempos em tempos, será bem simples e rápido e nos ajudará a perceber as variações que ocorrem quando movimentamos o sensor (dependendo da quantidade de precisão

que você utilize, ocorrerão pequenas variações mesmo com o dispositivo parado).

A primeira coisa a fazer é criar um novo projeto WPF no visual studio, como já fizemos no capítulo 2. Os passos para configurar o sensor serão muito simples, basta criar uma referência para nossa DLL auxiliar e utilizar o método `InicializarPrimeiroSensor` descrito em 2.1. Desta vez não precisamos ativar nenhum outro recurso do Kinect, então poderemos simplesmente utilizar o método já criado no construtor da janela, conforme o código a seguir.

```
public MainWindow()
{
    InitializeComponent();
    kinect = InicializadorKinect.InicializarPrimeiroSensor(0);
}
```

Você pode notar que não estamos declarando o objeto `KinectSensor kinect` no escopo do construtor da janela, você deve declará-lo no escopo da classe, pois iremos reutilizá-lo em outro método. Feito isso, já temos novamente nossa inicialização do sensor, agora iremos criar os componentes de tela para exibir os valores do acelerômetro nos eixos X, Y e Z, ignoraremos o eixo W, pois como citado acima ele sempre estará em 0 (zero).

Ao criar um novo projeto WPF o formulário deve estar apenas com um painel do tipo `Grid`, vamos utilizar este mesmo painel para construir um layout em forma de tabela cuja primeira coluna irá listar os eixos e a segunda, o valor de cada um.

XAML

Os projetos WPF utilizam o XAML *(eXtensible Application Markup Language)* para a construção da interface, esta é uma linguagem de marcação simples que possui foco em aplicações desktop com um papel similar ao HTML e CSS para aplicações Web.

Para que esta janela ocupe apenas o espaço necessário, iremos diminuir o seu tamanho, para isso vamos alterar os valores das propriedades `Width` e `Height` para os valores 300 e 150 respectivamente. Com isso, a janela ficou bem pequena, mas é suficiente para mostrarmos essas informações.

Feito isso, já podemos escolher o tamanho das linhas e colunas para exibir as informações. Como já dito antes iremos mostrar os valores dos eixos X, Y e Z, por-

tanto precisamos de três linhas do mesmo tamanho. Para fazer isso você deve criar a definição das linhas, conforme o código a seguir.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
</Grid>
```

Esta é a sintaxe para a criação das definições de linhas. Note que a propriedade `Height` está com o valor (*), este valor indica proporcionalidade, ou seja, todas as linhas ficarão com o mesmo tamanho — caso uma linha receba o valor (**2***), ela teria o dobro do tamanho das outras.

Agora iremos definir as duas colunas. A sintaxe é muito similar, mas ao invés de utilizarmos a propriedade `Height` que indica altura, iremos utilizar a propriedade `Width` que indica largura. Além disso, as colunas não irão possuir os mesmos valores proporcionalmente, pois a primeira irá exibir apenas uma letra referente ao seu eixo e a segunda irá receber um número com diversas casas decimais. O tamanho da primeira coluna será fixo em 50 pixels enquanto que a segunda irá ocupar todo o restante da janela, o código a seguir mostra como a tag `Grid` ficou após estas definições.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="50"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
</Grid>
```

Nossa estrutura do layout já está pronta! Agora vamos incluir os componentes para exibir as informações. Um componente simples e útil para exibir informações é o campo `Label`. Teremos 6 campos deste tipo e iremos definir algumas propriedades visuais para eles. Os três primeiros campos devem receber os valores X, Y

e Z na propriedade `Content` e os três últimos campos devem receber os valores labelX, labelY e labelZ na propriedade `Name` para que eles se tornem acessíveis no *codebehind* de nossa janela. Precisamos disso porque vamos atualizar o valor da propriedade `Content` com o valor do acelerômetro em nosso código C#, além disso, é importante utilizar as propriedades de alinhamento e as propriedades herdadas do componente `Grid` para posicionar a informação adequadamente no nosso layout. Com tudo terminado, o código desta janela deve ficar semelhante ao código a seguir.

```
<Window x:Class="Acelerometro.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="150" Width="300">

    <Grid>

        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="50"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>

            <Label Content="X" HorizontalAlignment="Center"
                   FontSize="24" VerticalAlignment="Center" />
            <Label Content="Y" HorizontalAlignment="Center"
                   FontSize="24" VerticalAlignment="Center"
                   Grid.Row="1"/>
            <Label Content="Z" HorizontalAlignment="Center"
                   FontSize="24" VerticalAlignment="Center"
                   Grid.Row="2"/>

            <Label Name="labelX" HorizontalAlignment="Left"
                   FontSize="24" VerticalAlignment="Center"
                   Margin="10,0" Grid.Column="1"   />
            <Label Name="labelY" HorizontalAlignment="Left"
                   FontSize="24" VerticalAlignment="Center"
                   Margin="10,0" Grid.Column="1" Grid.Row="1"/>
```

```
            <Label Name="labelZ" HorizontalAlignment="Left"
                   FontSize="24" VerticalAlignment="Center"
                   Margin="10,0" Grid.Column="1" Grid.Row="2"/>

    </Grid>
</Window>
```

Agora nossa janela já está pronta. Podemos implementar o código que irá atualizar estes campos com os valores do acelerômetro do sensor. Para fazer isso criaremos um método chamado `AtualizarValoresAcelerometro()`, no qual iremos obter os valores através do método disponível na SDK e inseri-los na propriedade `Content` dos componentes labelX, labelY e labelZ, conforme o código a seguir.

```
private void AtualizarValoresAcelerometro()
{
    Vector4 resultado = kinect.AccelerometerGetCurrentReading();
    labelX.Content = Math.Round(resultado.X, 3);
    labelY.Content = Math.Round(resultado.Y, 3);
    labelZ.Content = Math.Round(resultado.Z, 3);
}
```

Esta é a forma de obter os valores do acelerômetro, bem simples não é? Agora temos que observar um ponto importante, quando este método será executado? Conforme dito anteriormente não há eventos para a atualização do acelerômetro, pois ele sofre variações constantemente, para atualizarmos o valor iremos criar nosso próprio *timer*.

O objeto `DispatcherTimer` funciona como um cronômetro: cada vez que o tempo definido como seu intervalo passa, o evento `Tick` é chamado. Vamos interpretar este evento e dentro dele chamaremos o método criado anteriormente. Para criar um *timer* em C# basta utilizar o objeto mencionado, preencher a propriedade `Interval` que irá definir o intervalo com que o evento `Tick` é chamado e invocar o método `Start` para que o *timer* inicie. Veja como o método de inicialização do *timer* e o método que interpreta o evento `Tick` ficaram simples.

```
private void InicializarTimer()
{
    DispatcherTimer timer = new DispatcherTimer();
    timer.Interval = TimeSpan.FromMilliseconds(100);
    timer.Tick += timer_Tick;
    timer.Start();
```

```
}

private void timer_Tick(object sender, EventArgs e)
{
    AtualizarValoresAcelerometro();
}
```

Agora basta que o construtor de nossa janela invoque os métodos para inicializar o sensor Kinect e o *timer* que tudo irá funcionar. Execute sua aplicação e veja os valores do acelerômetro sendo alterados conforme você mexe no sensor. A figura 3.1 ilustra a janela da nossa aplicação.
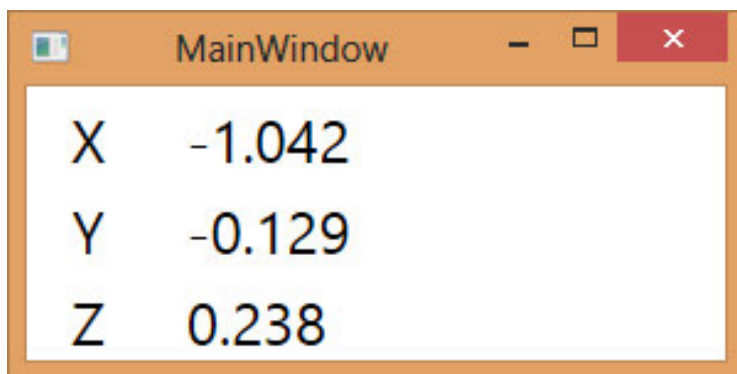
Figura 3.1: Aplicação Acelerômetro

3.2 Eixo Motorizado

O Eixo motorizado é uma peça importante no Kinect, através dele é possível alterar o ângulo de elevação do componente onde ficam as câmeras do sensor, alterando assim a visão que o sensor possui do ambiente. Este eixo movimenta o sensor apenas para cima e para baixo, atualmente não há uma forma de movimentá-lo horizontalmente.

O Kinect possui um raio de visão de 57.5 graus na horizontal e 43.5 graus na vertical, contudo este eixo motorizado pode movimentar a visão vertical para 27 graus para cima ou para baixo, conforme ilustrado na figura 3.2.
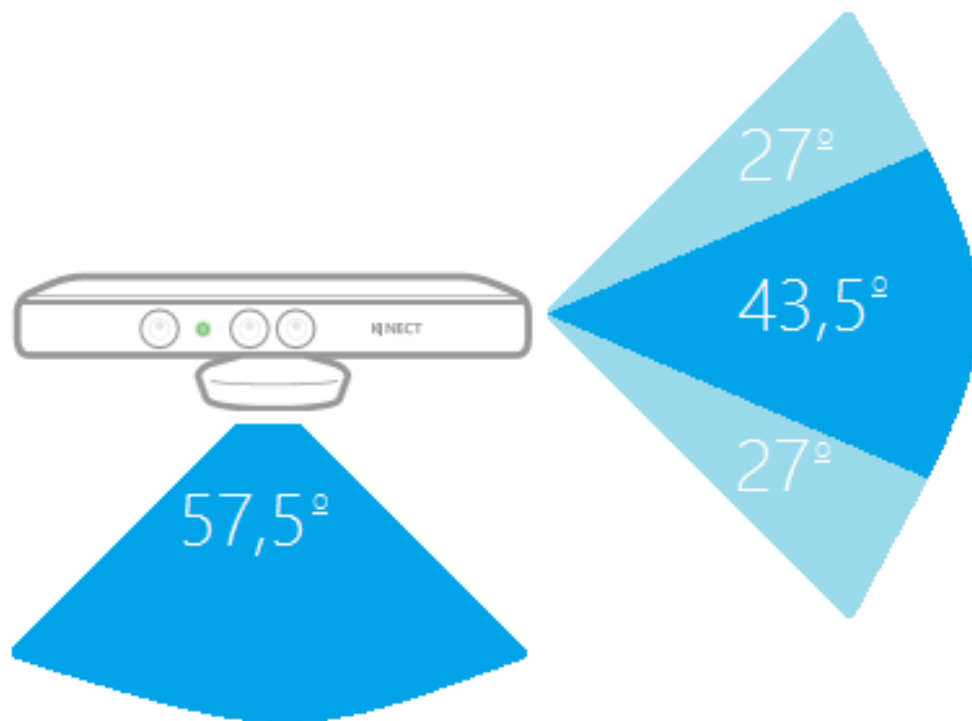
Figura 3.2: Visão do Kinect

As informações ligadas ao eixo motorizado são disponibilizadas no SDK de forma bastante simples.

Existem três propriedades na classe `KinectSensor`: `ElevationAngle`, `MaxElevationAngle` e `MinElevationAngle`. As duas últimas citadas possuem o valor fixo de 27 e -27 respectivamente. A propriedade `ElevationAngle` possui o valor igual ao valor da altura do ângulo de elevação do eixo motorizado. Nós já a utilizamos nos exercícios anteriores e você viu o quanto é simples alterar ou obter este valor.

PROBLEMA COM O ELEVATIONANGLE

Caso uma segunda alteração na propriedade ElevationAngle seja feita sem que a primeira tenha sido finalizada é lançada uma exceção na aplicação.

Aplicação

Construiremos outra aplicação WPF, então você deve fazer os passos já conhecidos para a criação do projeto, incluindo a inclusão e chamada para a inicialização do sensor descrita em 2.1. Nesta aplicação iremos construir um componente que nos permitirá alterar o valor do ângulo de elevação do eixo motorizado pela própria interface!

Como na aplicação anterior também utilizaremos o componente de layout `Grid`, dessa vez precisamos apenas de duas colunas, sendo que a segunda irá ocupar o dobro de espaço da primeira. Para fazer isso, utilize o caractere de proporção (*). Na primeira coluna da janela inserimos um componente `Slider`, que irá se encarregar de alterar a propriedade `ElevationAngle` e na outra coluna vamos inserir um componente `Label` para mostrar o valor do ângulo de elevação do Kinect.

É importante lembrar que precisamos inserir como valor máximo e mínimo do `Slider` 27 e -27 respectivamente, pois este é o valor limite do ângulo de elevação. Por opção irei inicializar a propriedade `Value` do `Slider` com o valor 0, ou seja, com o Kinect centralizado. Através do XAML, também podemos interpretar os eventos dos componentes: como teremos que atualizar o valor da propriedade `ElevationAngle` toda vez que o `Slider` tiver seu valor alterado, temos que interpretar o evento `Thumb.DragCompleted`, que será chamado toda vez que o usuário interagir com o componente.

Também precisamos preencher a propriedade `Name` dos dois componentes, pois iremos utilizá-los no código C# da janela. Após as configurações de layout feitas o código da janela deve estar semelhante ao código a seguir.

```
<Window x:Class="EixoMotorizado.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="230" Width="180">
    <Grid>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>

    <Slider Name="slider" Margin="10,0" Height="150"
    HorizontalAlignment="Left" VerticalAlignment="Center"
    Orientation="Vertical" Minimum="-27" Maximum="27"
```

```
    Value="0" Grid.Column="0"
    Thumb.DragCompleted="slider_DragCompleted"/>

    <Label Name="label" FontSize="64"
    FontFamily="Segoe UI Light"
    HorizontalAlignment="Center" VerticalAlignment="Center"
    Content="0" Grid.Column="1" />

    </Grid>
</Window>
```

Tendo a janela pronta, podemos partir para a implementação em C# desta aplicação. Criaremos um método dentro da nossa janela para atualizar os valores da propriedade `ElevationAngle` do sensor e do texto exibido no label. Chamaremos este método de `AtualizarValores` e seu código deve ficar similar ao código a seguir.

```
private void AtualizarValores()
{
    kinect.ElevationAngle = Convert.ToInt32(slider.Value);
    label.Content = kinect.ElevationAngle;
}
```

Simples, não é mesmo? Agora basta criar uma chamada para este método dentro do método `slider_DragCompleted` e tudo já irá funcionar!
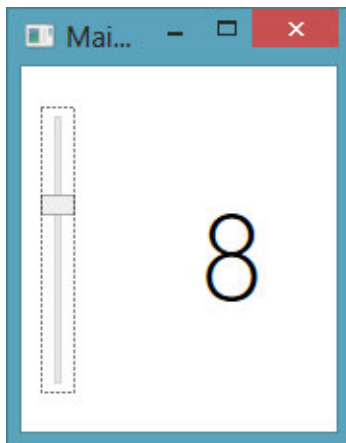
Figura 3.3: Aplicação EixoMotorizado

Capítulo 4

Fluxo de Cores

Nos três capítulos seguintes serão apresentadas as funcionalidades e processamentos que geram os fluxos (*streams*) que fazem parte da classe `KinectSensor`. Eles são: fluxo de cores (`ColorStream`), fluxo de profundidade (`DepthStream`) e fluxo de esqueleto de usuário(`SkeletonStream`). Todos possuem algumas características em comum e estão ligados a um ou mais sensores que envolvem seu processamento. Ambos precisam ser ligados para que o Kinect comece a processá-los e possuem dois caminhos para se obter um quadro, ou utilizando um método para solicitar o último quadro já processado, ou através de um evento que é disparado cada vez que um novo quadro deste fluxo está pronto, ou seja, já completou sua etapa de processamento. Além disso, podem ser acessados através do evento `AllFramesReady`, que possui informações destes três tipos de fluxos.

O fluxograma de funcionamento de todos os fluxos supracitados é ilustrado pela figura 4.1.
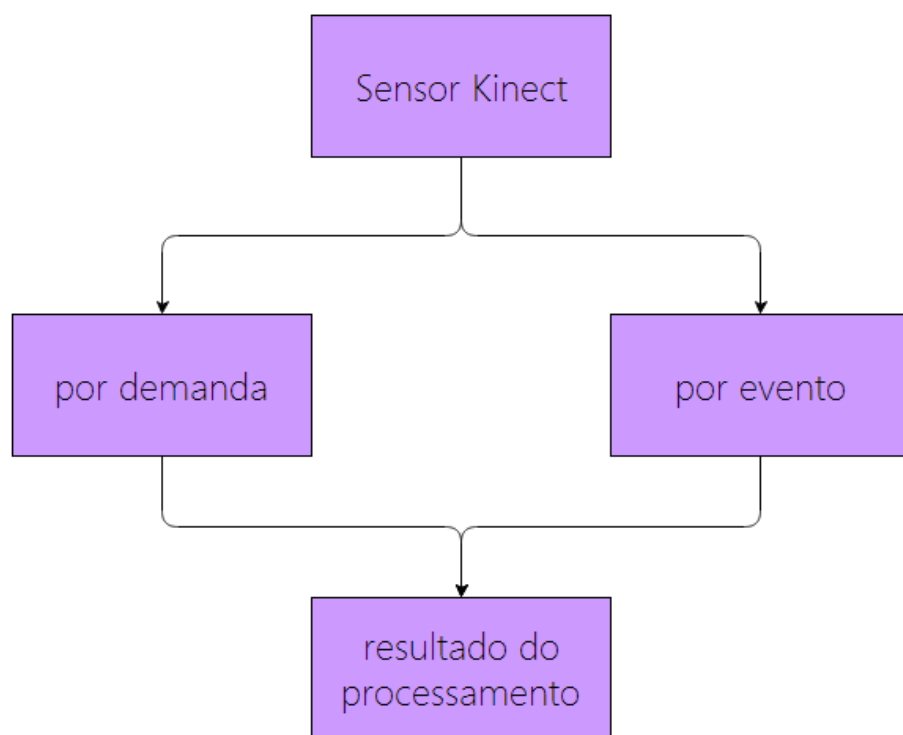
Figura 4.1: Fluxograma de processamento

Neste capítulo vamos entender como funciona a câmera de cores do Kinect, como utilizá-la para capturar fotos e vídeos, como tratar as imagens bit por bit e como aplicar estas funcionalidades em uma aplicação real.

4.1 Formatos

A câmera de cores do Kinect possui uma série de configurações, que influência diretamente na qualidade da imagem e na quantidade de quadros por segundo que o Kinect consegue processar. A tabela 4.2 ilustra estas resoluções.

Formatos	RGB		YUV	Bayer		IR
Bits por pixel	32		16	32		16
Resolução do eixo X	640	1280	640	640	1280	640
Resolução do eixo Y	480	960	480	480	960	480
Quadros por segundo	30	12	12	30	12	12

Figura 4.2: Tabela de Configurações da Câmera RGB

Você provavelmente já ouviu falar sobre o formato RGB ou (Red-Green-Blue), bastante conhecido no ramo da computação. Neste formato, cada pixel da imagem possui uma quantidade da cor vermelha, uma quantidade da cor verde e uma quantidade da cor azul, a soma destes três valores representam a cor do pixel.

O formato YUV codifica uma imagem ou video levando em conta a percepção humana. Ele permite uma redução da largura de banda de uma transmissão pois utiliza somente 16 bits por pixel, então é muito útil para comprimir imagens, pois a perda é mascarada pela percepção humana, enquanto que o RGB possui o dobro de tamanho por pixel e nem sempre faz diferença para nossos olhos. Na sigla YUV, o Y representa a luminância de um pixel, ou seja, quantidade de luminosidade percebida pelo olho humano naquele pixel, enquanto que as siglas U e V representam dois componentes diferentes de crominância, que é a forma como nossos olhos percebem as cores.

A figura 4.3 ilustra a imagem com os componentes separados.
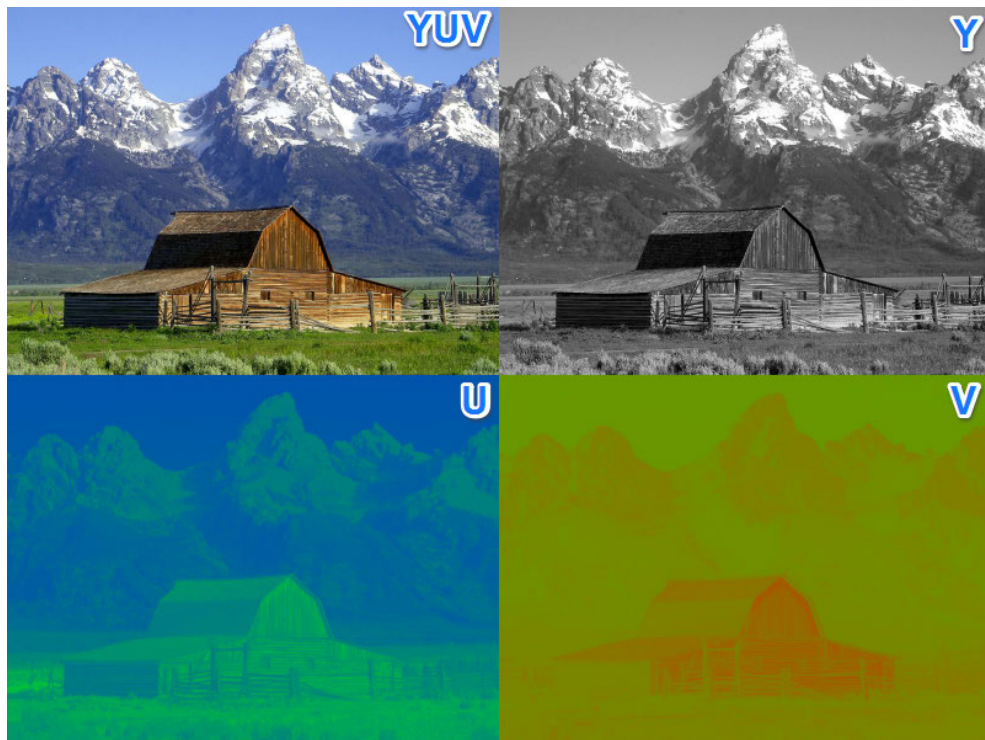
Figura 4.3: Imagem YUV

O formato Bayer é bastante similar ao RGB, mas ele altera a imagem proporcionalmente à nossa percepção para cores, então o verde (que é a cor que mais percebemos) é mais levado em consideração do que o vermelho ou azul. As diferenças visuais entre os formatos RGB, YUV e Bayer são bem sutis, a figura 4.4 ilustra uma mesma imagem nos três formatos com a resolução máxima.
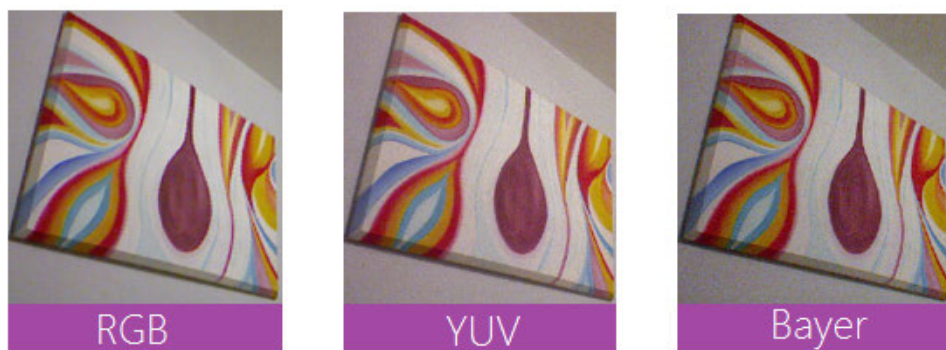
Figura 4.4: Comparação entre formatos da Câmera RGB

O formato *InfraRed* ou infravermelho é bem diferente dos citados anteriormente. Quando a câmera é habilitada com este formato, podemos visualizar a forma com que o Kinect "vê" o ambiente. O resultado é uma imagem escura com diversos pontos luminosos, que representam os pontos infravermelhos que são lançados pelo emissor de luz infravermelho no ambiente.

4.2 Aplicação

Neste capítulo iremos criar uma aplicação um pouco mais complexa, que contará com as funções mencionadas anteriormente: bater uma foto e exibir na aplicação a imagem (que será feito utilizando o fluxo de cores sob demanda), exibir na aplicação um o vídeo do que a câmera RGB está vendo (que será feito utilizando o evento) e criaremos uma opção para utilizar um filtro de escala cinza sobre a imagem.

Como você já deve estar acostumado criaremos uma nova aplicação WPF e novamente iremos incluir as referências para nosso projeto auxiliar e para a SDK do Kinect. Nossa aplicação será construída por etapas — primeiramente iremos fazer com que ela apenas bata foto sob demanda.

Criando o layout

O Layout desta aplicação será relativamente simples. Primeiro vamos dividir o componente `Grid` em duas linhas, utilizando o `Grid.RowDefinitions` que já mencionamos anteriormente. A linha inferior não deve receber um valor muito alto

para sua altura (50 é suficiente), pois ela irá servir apenas para inserirmos nosso botão que baterá a foto, todo o resto da altura da janela deve pertencer a linha superior, já que ela terá o componente que irá exibir a imagem de nossa foto. O código de sua tela deve ficar similar ao código a seguir.

```
<Window x:Class="SensorRGB.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="50"/>
        </Grid.RowDefinitions>

        <Button Content="Bater Foto" Grid.Row="1" Margin="10,10"
        Width="70" HorizontalAlignment="Left"
        Click="Button_BaterFoto"/>

        <Image Name="imagemCamera" />

    </Grid>
</Window>
```

Utilizando um quadro sob demanda

Note que, no código anterior, já foi atribuído um evento para o botão que irá bater a foto, vamos então ao código. Como padrão, após o método `InitializeComponent` da janela devemos inicializar o sensor de nossa aplicação. No caso desta aplicação, iremos criar um método chamado `InicializarKinect()`, pois além de fazermos a inicialização padrão temos que inicializar o fluxo de cores. Para isso, basta utilizarmos o método `Enable()` do objeto `ColorStream` que pertence ao sensor, ele pode receber por parâmetro o formato que a câmera irá utilizar (RGB, YUV, Bayer ou IR). Caso você não informe nada ele assumirá o formato padrão (`ColorImageFormat.RgbResolution640x480Fps30`).

O corpo do método criado deve ficar similar ao código a seguir.

```
private void InicializarKinect()
{
```

```
    kinect = InicializadorKinect.InicializarPrimeiroSensor(10);
    kinect.ColorStream.Enable();
}
```

Neste caso optei por levantar o eixo do sensor 10 graus, mas isso pode ficar a seu critério. Você pode notar que a inicialização dos fluxos do Kinect pode ser feita de maneira bastante intuitiva e simples. Essa forma é a mesma para os três tipos de fluxo citados anteriormente.

Agora iremos criar o método que irá receber um quadro de cores do sensor RGB e irá obter a imagem deste quadro. Você notará que apesar de parecer complicado, também é uma tarefa bastante simples. Este método irá se chamar `ObterImagemSensorRGB`, ele deve retornar um `ImageSource` que será inserido como fonte de dados para o componente de imagem e receberá por parâmetro um objeto do tipo `ColorImageFrame`, que representa um quadro do sensor RGB.

Um objeto do tipo `ColorImageFrame` é um objeto que pode ser classificado como recurso, ele implementa a interface `IDisposable`, ou seja, você deve utilizar o método `Dispose()` para destruir o objeto ou simplesmente utilizar o comando `using`. Para tornar o código mais legível, utilizaremos no exemplo a segunda opção. Dentro do escopo do bloco `using` teremos que criar um array de bytes e o converteremos para a fonte do componente de imagem.

```
private BitmapSource ObterImagemSensorRGB(ColorImageFrame quadro)
{
    using (quadro)
    {
        byte[] bytesImagem = new byte[quadro.PixelDataLength];
        quadro.CopyPixelDataTo(bytesImagem);

        return BitmapSource.Create(quadro.Width, quadro.Height,
        96, 96, PixelFormats.Bgr32, null, bytesImagem,
        quadro.Width * quadro.BytesPerPixel);

    }
}
```

Como você pode ver no método anterior, praticamente toda a informação referente à imagem é retirada do objeto `quadro`. O método estático `Create` da classe `BitmapSource` recebe uma série de parâmetros, agora iremos entender o significado de cada um deles.

Os dois primeiros são respectivamente a largura e a altura da imagem, basta retirar estas informações do quadro. Os dois seguintes que estão no exemplo: 96,96 são a quantidade de pontos por polegadas (dpi) nos eixos X e Y. O parâmetro seguinte se refere ao formato de cada pixel da imagem — caso você altere o formato de inicialização do fluxo de cores possivelmente também terá de alterar o formato dos pixels da imagem. O parâmetro que está sendo enviado como `null` é utilizado para passar uma possível palete de cores, o que neste caso não é interessante. Após este, são enviados por parâmetro os bytes que a imagem contém e finalmente por último é enviado o *stride* da imagem.

STRIDE

Quando uma imagem ou vídeo é armazenado na memória de um computador, seu buffer equivalente na memória pode conter informações extras, além da própria imagem após o fim de cada linha de pixels. Estas informações extras geralmente são utilizadas para afetar a forma como a imagem é armazenada na memória e não têm a ver com a forma com que a imagem é exibida.

http://msdn.microsoft.com/en-us/library/windows/desktop/aa473780

Após ter feito este método, precisamos invocá-lo no evento que interpreta o clique do botão. Ele irá conter apenas uma linha que fará com que a propriedade `Source` de nosso componente de imagem receba o retorno do método `ObterImagemSensorRGB`. Lembre-se que este último método citado precisa receber por parâmetro o quadro atual do sensor RGB. Para fazermos isso, utilizamos o método `OpenNextFrame` do objeto `ColorStream` do sensor, que requer um parâmetro que representa o tempo de espera (em milissegundos) até que o frame seja aberto. Neste exemplo utilizaremos o valor 0 (zero), pois desejamos que a foto seja tirada instantaneamente.

```
private void Button_BaterFoto(object sender, RoutedEventArgs e)
{
    imagemCamera.Source =
    ObterImagemSensorRGB(kinect.ColorStream.OpenNextFrame(0));
}
```

Agora podemos executar a aplicação e ao clicar no botão “Bater Foto” já temos o resultado esperado!
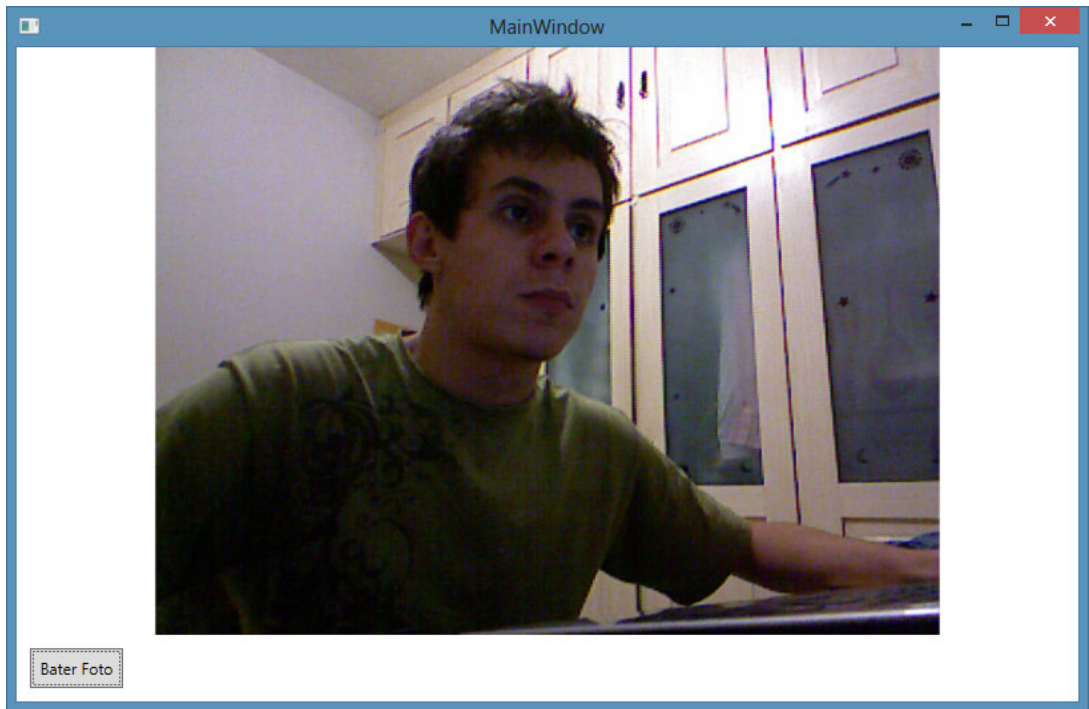
Figura 4.5: Aplicação utilizando o sensor RGB

Utilizando o evento de quadros novos em um fluxo de cores

Já vimos como podemos buscar quadros dos fluxos sob demanda, que tal refatorar esta implementação para utilizar o evento? Com isso teremos uma nova imagem a cada quadro, ou seja, 30 vezes por segundo. Primeiramente removemos o botão para bater foto (e seu método relacionado ao evento `Click`) de nossa aplicação, já que ele não faz mais sentido. Depois disso faremos um pequeno ajuste no método `InicializarKinect`, iremos interpretar o evento `ColorFrameReady` do objeto `ColorStream` após inicializar o fluxo deste objeto.

O método que interpretará este evento recebe por parâmetro um objeto do tipo `ColorImageFrameReadyEventArgs` e é deste objeto que teremos que retirar o novo quadro de imagem utilizando o método `OpenColorImageFrame`, ou seja,

faremos uma chamada muito similar à que existia no método do botão de bater foto, porém desta vez, utilizaremos o quadro do objeto que é enviado por parâmetro e não diretamente do objeto de fluxo.

```
private void kinect_ColorFrameReady(object sender,
ColorImageFrameReadyEventArgs e)
{
    imagemCamera.Source =
    ObterImagemSensorRGB(e.OpenColorImageFrame() );
}
```

Caso o Kinect leve mais tempo para inicializar que a primeira chamada do método descrito anteriormente, a aplicação lançará uma exceção, então, antes do comando `using` no método `ObterImagemSensorRGB` é interessante fazer uma validação do quadro atual, ou seja, verificar se o quadro não está nulo.

Se você fez tudo certo e executar a aplicação já irá ver o vídeo em tempo real do que o Kinect está vendo!

Utilizando um filtro de imagem

Que tal incrementar ainda mais nossa aplicação utilizando um filtro de imagem? Iremos fazer o exemplo utilizando um filtro simples para deixar a imagem em escala cinza (popularmente chamado de preto e branco). Primeiro faremos com que o filtro para escala cinza seja opcional, então criaremos um componente do tipo `CheckBox` no mesmo local onde ficava o botão para bater foto e este componente irá nos indicar quando deveremos aplicar o filtro.

Para aplicarmos o filtro, é preciso refatorar o método `ObterImagemSensorRGB`. Como você deve ter percebido nós podemos manipular cada pixel da imagem do quadro atual, então vamos utilizar um algoritmo simples para converter as cores RGB deste pixel em escala cinza.

Após copiarmos os dados da imagem para o array de pixel `byte[] bytesImagem`, faremos um laço de repetição para percorrer todos os pixels substituindo o valor das cores R, G e B pelo valor mais alto dos três (utilize o método `Math.Max`). Isso será feito porque toda a escala cinza de cores possui os valores RGB idênticos.

É importante ter em mente nesta implementação que um pixel não é necessariamente um byte. Dependendo do formato de cores são necessários mais de um byte para preencher um pixel — esta informação pode ser obtida através da propriedade

BytesPerPixel do objeto ColorImageFrame. No caso do formato padrão, cada pixel é representado por 4 bytes, um para o R, um para o G, um para o B e um para o Alpha (transparência), sendo assim, nosso laço de repetição deve pular de 4 em 4 bytes. Caso você tenha tentado refatorar o método, ele deve estar semelhante ao código a seguir.

```
private BitmapSource ObterImagemSensorRGB(ColorImageFrame quadro)
{
    if (quadro == null) return null;

    using (quadro)
    {
        byte[] bytesImagem = new byte[quadro.PixelDataLength];
        quadro.CopyPixelDataTo(bytesImagem);

        if (chkEscalaCinza.IsChecked.HasValue &&
            chkEscalaCinza.IsChecked.Value)

            for (int indice = 0;
                 indice < bytesImagem.Length;
                 indice += quadro.BytesPerPixel)
            {
                byte maiorValorCor = Math.Max(bytesImagem[indice],
                                     Math.Max(bytesImagem[indice + 1],
                                     bytesImagem[indice + 2]));

                bytesImagem[indice] = maiorValorCor;
                bytesImagem[indice + 1] = maiorValorCor;
                bytesImagem[indice + 2] = maiorValorCor;
            }

        return BitmapSource.Create(quadro.Width, quadro.Height,
        96, 96, PixelFormats.Bgr32, null, bytesImagem,
        quadro.Width * quadro.BytesPerPixel);
    }
}
```

O componente de tela no meu caso se chama chkEscalaCinza e ele é utilizado na verificação antes do laço de repetição. Com isso, já podemos executar nosso código e ao marcar o CheckBox teremos um resultado semelhante ao da figura 4.6.
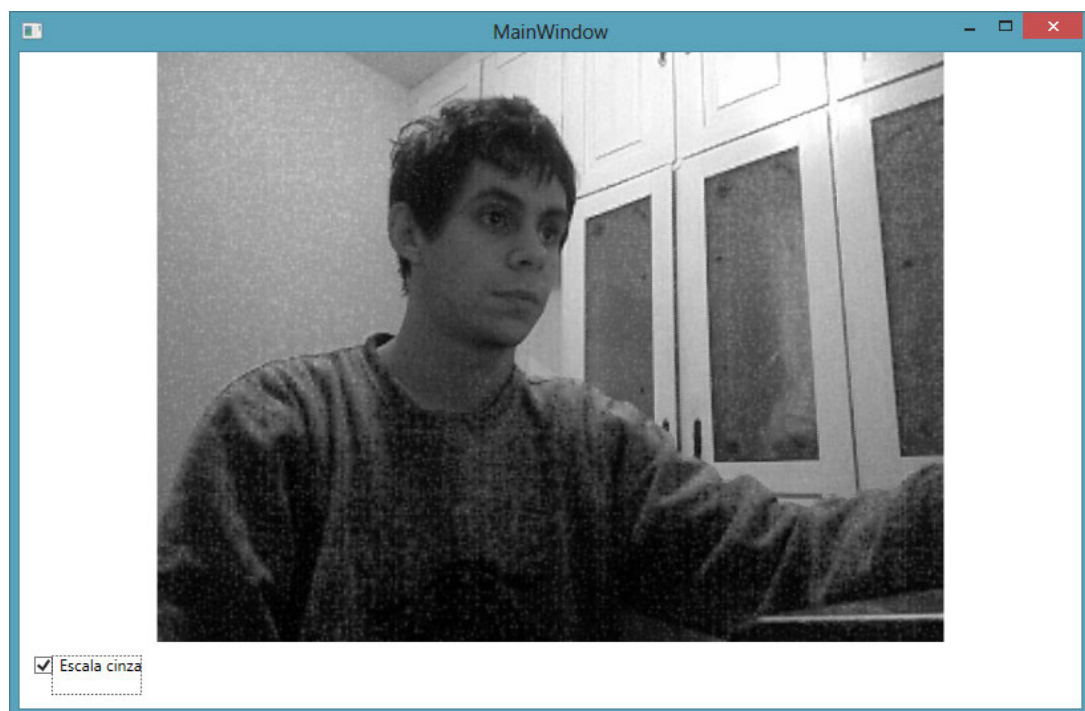
Figura 4.6: Filtro para escala cinza

4.3 Refatoração

Você deve se lembrar que na seção 1.3 havia sido mencionado o Kinect toolkit; pois bem, utilizaremos o toolkit (Microsoft.Kinect.Toolkit, Microsoft.Kinect.Toolkit.Controls e Microsoft.Kinect.Toolkit.Controls.Interaction) para fazer uma refatoração em nosso código.

Refatorando a DLL AuxiliarKinect

Primeiro vamos refatorar nosso projeto **AuxiliarKinect**, você se lembra que o método de inicialização não prevê nenhum tipo de problema? Bom, já está na hora de resolvermos isso. Primeiro adicione a referência somente à DLL Microsoft.Kinect.Toolkit. Nesta DLL há uma classe chamada `KinectSensorChooser`, que é feita para ser utilizada como um seletor do sensor Kinect e ela mesma já gerencia as possíveis exceções.

Refatoração

Nesta seção iremos refatorar a DLL AuxiliarKinect, então caso você queira manter as implementações antigas sugiro que faça uma cópia deste projeto antes de efetuar estas novas alterações.

A primeira coisa que faremos na nossa classe `InicializadorKinect` é criar uma propriedade da classe `KinectSensorChooser` chamada `SeletorKinect`. Sugiro que esta propriedade tenha o método `get` público e o método `set` privado.

Também criaremos uma propriedade totalmente pública neste classe do tipo `Action<KinectSensor>`. Esta propriedade é um delegate para a função que precisará ser invocada sempre que o sensor for inicializado, falaremos mais sobre ela mais tarde.

Criaremos também um construtor para nossa classe, que deve instanciar a propriedade `SeletorKinect`, associar um método que interprete o evento `KinectChanged` e inicializar o seletor, conforme o código a seguir.

```
public Action<KinectSensor> MetodoInicializadorKinect
{ get; set; }

public KinectSensorChooser SeletorKinect
{ get; private set; }

public InicializadorKinect()
{
    SeletorKinect = new KinectSensorChooser();
    SeletorKinect.KinectChanged += SeletorKinect_KinectChanged;
    SeletorKinect.Start();
}
```

O método `SeletorKinect_KinectChanged` é invocado toda vez que um sensor é conectado ou desconectado (por qualquer motivo) e as informações referentes a este sensor estão encapsuladas nas propriedades `OldSensor` e `NewSensor` do objeto `KinectChangedArgs` que é recebido por parâmetro. Nesta aplicação, iremos invocar o método que inicializa o sensor cada vez que um novo sensor é descoberto e desligar as funções de um sensor que foi desconectado, apenas para segurança, pois não é em todos os casos que o sensor foi desconectado da energia

— mesmo estando em um estado inválido ele ainda pode consumir recurso da máquina.

Para desligar todos os fluxos utilizaremos o método `Disable()` de cada fluxo, mas antes de desligá-los é necessário verificar se os mesmos estão ligados através da propriedade `IsEnabled`. É necessário que este desligamento esteja sendo feito dentro de um bloco `try catch`, pois o sensor pode entrar em um estado inválido durante alguma operação.

```
private void SeletorKinect_KinectChanged(object sender,
KinectChangedEventArgs kinectArgs)
{
    if (kinectArgs.OldSensor != null)
    {
        try
        {
            if (kinectArgs.OldSensor.DepthStream.IsEnabled)
                kinectArgs.OldSensor.DepthStream.Disable();

            if (kinectArgs.OldSensor.SkeletonStream.IsEnabled)
                kinectArgs.OldSensor.SkeletonStream.Disable();

            if (kinectArgs.OldSensor.ColorStream.IsEnabled)
                kinectArgs.OldSensor.ColorStream.Disable();
        }
        catch (InvalidOperationException)
        {
            // Captura exceção caso o KinectSensor entre
            // em um estado inválido durante a desabilitação
            // de um fluxo.
        }
    }

    if (kinectArgs.NewSensor != null)
    {
        if (MetodoInicializadorKinect != null)
            MetodoInicializadorKinect(SeletorKinect.Kinect);
    }

}
```

Refatorando a Aplicação SensorRGB

Agora iremos refatorar nossa aplicação para se adequar a este novo `AuxiliarKinect`. Basicamente o que precisamos fazer é alterar o método `InicializarKinect` para que, ao invés de utilizar o método estático de antes, passe a utilizar nosso novo seletor.

Antes de alterarmos este método, criaremos um novo método nesta aplicação, chamado `InicializarSeletor`. Ele irá instanciar um novo objeto `InicializadorKinect` e invocar o método já existente `InicializarKinect` que agora não deve mais conter a chamada para o antigo método estático da classe `InicializadorKinect` e deve receber o objeto Kinect do seletor por parâmetro. Seu código ficará similar ao código a seguir.

```
private void InicializarSeletor()
{
    InicializadorKinect inicializador = new InicializadorKinect();
    InicializarKinect(inicializador.SeletorKinect.Kinect);
}

private void InicializarKinect(KinectSensor kinectSensor)
{
    kinect = kinectSensor;
    kinect.ColorStream.Enable
            (ColorImageFormat.RgbResolution640x480Fps30);
    kinect.ColorFrameReady += kinect_ColorFrameReady;
}
```

Agora no construtor passaremos a invocar o método para inicializar o seletor e não mais o sensor, pois este método já será chamado internamente. Talvez você esteja com dúvidas quanto ao motivo de termos dois métodos separados, um para apenas o sensor e um para o seletor (que chama internamente o método do sensor), esta pergunta será respondida em breve.

Note que apenas duas linhas foram criadas, e será que a aplicação continua a mesma? A resposta é **não**, agora caso aconteça algum problema com o sensor a aplicação não dispara exceções não tratadas, e sim, apenas para de atualizar os quadros do vídeo. Porém ainda existe um problema não resolvido nesta refatoração: caso o sensor volte a seu estado funcional, ele não retornará a ligar o fluxo de cores e não voltará a atualizar a imagem. Faça o seguinte teste: remova seu sensor de seu computador (as imagens irão parar de ser atualizadas) e em seguida reconecte o sensor (nada deve acontecer).

Para resolver este problema, precisamos informar ao objeto `inicializador` que temos um método que é necessário executar sempre que um sensor for descoberto, para isso, utilizaremos a propriedade `MetodoInicializadorKinect`. Então no método `InicializarSeletor` após instanciar o objeto devemos atribuir à propriedade `MetodoInicializadorKinect` o método `InicializarKinect`. Feito isso não será mais necessário invocá-lo dentro do método que inicializa o seletor.

Refaça o teste de remover e reconectar o sensor em seu computador e você perceberá que agora ele volta a funcionar assim que o Kinect é reconectado! Mas ainda não acabamos, vamos à nossa janela novamente.

No `Grid` principal de nossa janela crie uma nova linha acima do painel de imagem com a mesma altura da linha onde está a marcação para escala cinza (não esqueça de incluir a propriedade `Grid.Row = "1"` no objeto `Image`). Agora usaremos o *Microsoft.Kinect.Toolkit.Controls.Interactions* para alterar a interface de nossa janela.

Após ter criado a nova linha no `Grid` vamos adicionar a referência ao *Toolkit.Controls.Interactions* dentro da própria janela na linguagem XAML. Procure pela linha `xmlns:x="`http://schemas.microsoft.com/winfx/2006/xaml" , esta é a sintaxe de uma referência para um schema em XAML. Abaixo dela, adicione a seguinte linha; `xmlns:k="`http://schemas.microsoft.com/kinect/2013" com isso já temos a referência para o namespace onde se encontra componente visual do seletor.

Vamos adicioná-lo à linha criada no `Grid` e iremos posicionar verticalmente no topo e horizontalmente no centro. Lembre-se que para acessar um componente em XAML que pertence a um namespace é necessário informar o namespace antes, utilizando a seguinte sintaxe: **namespace:componente**, seu XAML deve ter ficado similar ao código a seguir.

```
<Window x:Class="SensorRGB.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:k="http://schemas.microsoft.com/kinect/2013"
    Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="50"/>
            <RowDefinition Height="239*"/>
            <RowDefinition Height="50"/>
        </Grid.RowDefinitions>
```

```
        <Image Name="imagemCamera" Grid.Row="1" />

        <CheckBox Name="chkEscalaCinza" Content="Escala cinza"
        HorizontalAlignment="Left" Margin="10,10,0,10" Grid.Row="2"/>

        <k:KinectSensorChooserUI Name="seletorSensorUI"
        HorizontalAlignment="Center" VerticalAlignment="Top" />

    </Grid>
</Window>
```

Por último temos de ligar a nossa classe seletora ao nosso componente visual, iremos fazer isso após a última linha do método `InicializarSeletor`.

```
private void InicializarSeletor()
{
    InicializadorKinect inicializador = new InicializadorKinect();
    inicializador.MetodoInicializadorKinect = InicializarKinect;
    seletorSensorUI.KinectSensorChooser = inicializador.SeletorKinect;
}
```

Agora, ao executar novamente a aplicação você irá perceber o comportamento deste controle (ele exibe apenas um ícone, mas caso você passe o mouse acima do ícone é exibido um painel detalhado). Caso você desconecte o sensor como fizemos antes, ele irá exibir um alerta conforme figura 4.7 (a mensagem do alerta varia de acordo com o erro), e caso reconecte, ele mostra que o sensor está pronto novamente. Este componente é muito útil para criar uma interface agradável com o usuário e tornar sua aplicação mais responsiva.
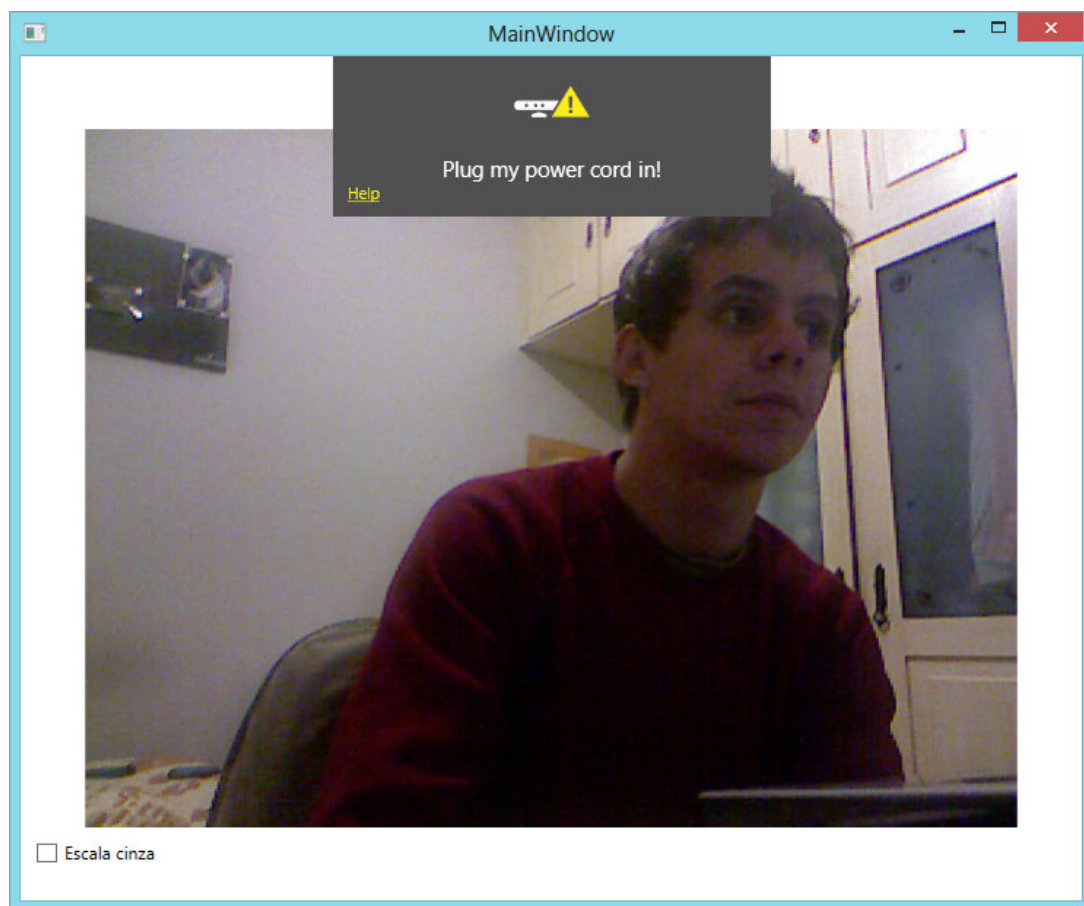
Figura 4.7: Seletor do Kinect

Que tal incrementarmos nossa aplicação utilizando o que aprendemos no capítulo anterior? Vamos inserir um componente `Slider` para podermos alterar o ângulo de elevação do sensor nesta aplicação. Desta vez iremos fazer os passos um pouco mais rápido, pois já fizemos uma implementação parecida.

Primeiro iremos alterar o layout de nossa aplicação, adicionando uma coluna ao `Grid` principal. Esta coluna será adicionada no lado direito da janela e deverá ter uma largura de tamanho 25. Nesta coluna iremos adicionar o componente `Slider` seguindo as mesmas configurações da aplicação anterior (Valor máximo e mínimo). Para fins estéticos iremos fazer um *Binding* na propriedade `Height` do componente `Slider` com a propriedade `ActualHeight` do componente `Image` do formulário.

Binding

Binding significa "fazer ligação". Este recurso geralmente é utilizado para fazer uma ligação entre um dado do banco de dados e um componente visual em uma janela, mas em aplicações WPF podemos utilizar esta técnica para vincular informações de qualquer tipo. No caso citado anteriormente, o valor da propriedade que representa a altura do componente `Slider` sempre possuirá o mesmo valor da propriedade que representa a altura da imagem buscada pelo Kinect, harmonizando o layout da aplicação.

Também para fins estéticos iremos alterar a propriedade `Stretch` do componente imagem para o valor `"Fill"`. Com isso, a imagem será "esticada" para ocupar todo espaço do componente. Após fazer estas implementações o arquivo XAML, sua janela deve estar parecida com o código a seguir.

```
<Window x:Class="SensorRGB.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:k="http://schemas.microsoft.com/kinect/2013"
    Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="25"/>
           </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="50"/>
            <RowDefinition Height="240*"/>
            <RowDefinition Height="50"/>
        </Grid.RowDefinitions>

        <Image Name="imagemCamera" Grid.Row="1" Stretch="Fill"/>

        <CheckBox Name="chkEscalaCinza" Content="Escala cinza"
        HorizontalAlignment="Left" Margin="10,10,0,10" Grid.Row="2"/>

        <k:KinectSensorChooserUI Name="seletorSensorUI"
        HorizontalAlignment="Center" VerticalAlignment="Top" />
```

```
        <Slider Name="slider" Width="20" Orientation="Vertical"
        Minimum="-27" Maximum="27" SmallChange="1" Value="0"
        Height="{Binding ElementName=imagemCamera, Path=ActualHeight}"
        Thumb.DragCompleted="slider_DragCompleted"
        Grid.Column="1" Grid.Row="1"/>

    </Grid>
</Window>
```

Como você pode ter notado, no código anterior o componente nomeado slider, interpreta o evento `DragCompleted`. Este é o mesmo evento que utilizamos no capítulo anterior, agora temos de implementá-lo para que ele sincronize a propriedade `ElevationAngle` do Kinect com o valor do componente slider.

```
private void slider_DragCompleted
(object sender,
    System.Windows.Controls.Primitives.DragCompletedEventArgs e)
{
    kinect.ElevationAngle = Convert.ToInt32(slider.Value);
}
```

Após estas implementações você já pode utilizar a aplicação e alterar o ângulo de elevação do sensor através do componente visual!
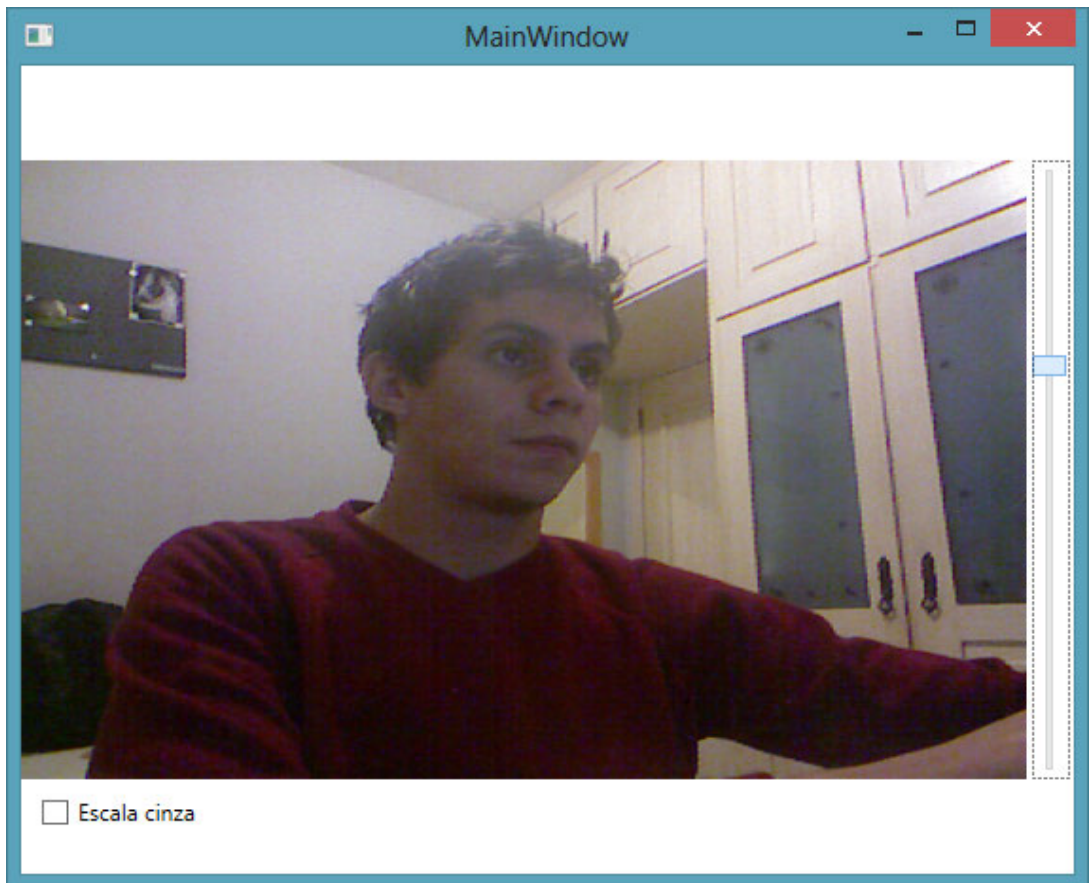
Figura 4.8: Aplicação com o componente Slider

Antes de passarmos para o novo assunto eu sugiro que você faça mais testes com outros tipos de formato para o fluxo de cores. Faremos uma pequena alteração para o formato *infrared* apenas para termos contato com este formato que é bem diferente dos demais.

Para alterar o formato do fluxo de cores basta passar por parâmetro no método `Enable` o formato `InfraredResolution640x480Fps30`, além disso, em nossa criação do `BitmapSource` precisamos alterar o `PixelFormat` para `Gray16`. Com estas simples alterações, a imagem que será exibida no componente já será totalmente diferente, conforme a figura 4.9.

Figura 4.9: Fluxo de cores infravermelho

É importante lembrar-se que o SDK não permite a inicialização de dois formatos diferentes para o mesmo fluxo, então não é possível habilitar o infravermelho e o RGB, por exemplo, visto que o infravermelho apesar de ser bastante diferente não é um fluxo a parte, e sim uma configuração do fluxo de cores.

Capítulo 5

Fluxo de Profundidade

Neste capítulo iremos incrementar a aplicação anterior utilizando alguns conceitos de profundidade. Veremos como utilizaremos as informações deste fluxo para detectarmos usuários e a distância que os objetos estão do sensor.

5.1 Formatos

O sensor de profundidade do Kinect também possui diferentes tipos de formato e para utilizar um formato é exatamente igual ao sensor RGB, ou seja, o parâmetro referente ao formato pode ser informado no método `Enable` do fluxo de profundidade (`DepthStream`).

No formato deste fluxo, é possível alterar somente a resolução, pois em todos os formatos o *FPS (frames per second)* ou quadros por segundo permanece 30 e a quantidade de bits por pixel permanece 16. A figura 5.1 ilustra a diferença entre os formatos.
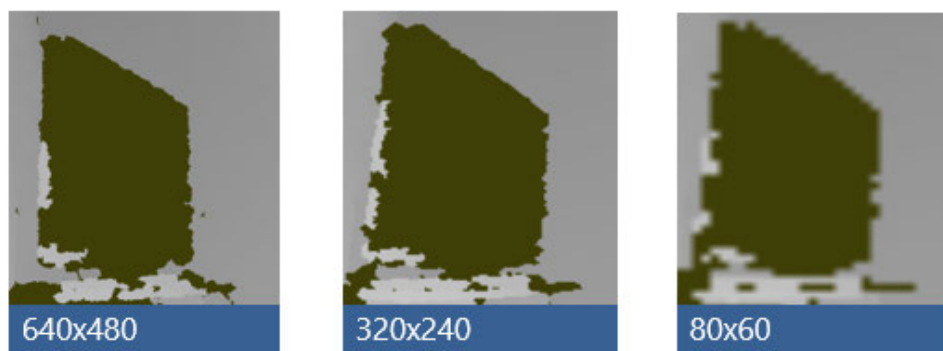
Figura 5.1: Tabela de Configurações do sensor de profundidade

5.2 Entendendo um pouco mais sobre a profundidade

A configuração de *near mode* disponível somente na versão do Kinect para Windows afeta diretamente este fluxo, pois o Kinect possui uma distância mínima e máxima para que ele possa verificar a distâncias dos objetos. Esta variação de distância no formato normal é de no mínimo 0,8m e no máximo 4m. No *near mode* esta distância muda para 0,5m no mínimo e no máximo 3m. Você deve optar pelo formato que mais se encaixa em sua aplicação, ou utilizar somente o modo padrão caso a versão de seu Kinect seja a versão para Xbox 360.

A imagem reconhecida pelo Kinect pode ser descrita em um formato conhecido como RGBD, ou seja, *Red-Green-Blue-Depth*. Todo pixel de profundidade possui 16 bits, sendo 13 bits para informações referentes à profundidade e 3 bits que identificam se o pixel pertence à um humano.

Esta área de 3 bits que identifica os humanos é conhecida por *player segmentation data*, ou seja, segmento de dados de um jogador ou usuário. A câmera de profundidade é capaz de reconhecer até 6 usuários em frente ao sensor e todas estas informações citadas são obtidas através da classe `DepthImagePixel`.

O fluxo de profundidade está intimamente ligado ao fluxo de esqueleto, pois ambos utilizam recursos uns dos outros para obterem informações, um exemplo disso é o próprio *player segmentation data*. Para que estas informações estejam disponíveis no quadro de profundidade é necessário que o fluxo de esqueleto esteja ativo.

Todas estas informações referentes à profundidade são capturadas pelo Kinect

utilizando uma técnica conhecida como **Efeito Parallax**. Este efeito ocorre naturalmente em nossa própria visão para identificar a distância dos objetos. Não cabe ao escopo deste livro a explicação deste efeito, mas para que você possa compreendê-lo de maneira simples e superficial imagine o seguinte cenário, você está em um carro em uma rodovia e ao lado desta rodovia existem diversas árvores e montanhas, ao olhar pela janela você consegue perceber que as árvores que estão mais próximas à rodovia passam mais rápido do que a montanha que está ao fundo, esta percepção que temos é o **efeito parallax** citado anteriormente.

O sensor utiliza este efeito, seus dois sensores de profundidade e um algoritmo para detectar a profundidade de cada objeto na imagem. Tanto o efeito parallax quanto a técnica utilizada pelo Kinect são mais complexos do que estas breves explicações, mas iremos focar na implicação disso em uma aplicação final e não no modo como o Kinect foi concebido.

5.3 Aplicação

Agora iremos por em prática os conceitos vistos anteriormente neste capítulo, utilizaremos a mesma aplicação do capítulo anterior, mas inicialmente vamos desabilitar o fluxo de cores e habilitar o fluxo de profundidade. O primeiro passo será alterar o método `InicializarKinect`, que informamos na propriedade `MetodoInicializadorKinect` de nosso inicializador.

Vamos comentar as linhas que inicia o fluxo de cores e substituí-las por linhas que iniciam o fluxo de profundidade e criam um método para interpretar seu evento, conforme código a seguir.

```
private void InicializarKinect(KinectSensor kinectSensor)
{
    kinect = kinectSensor;
    slider.Value = kinect.ElevationAngle;

    kinect.DepthStream.Enable();
    kinect.DepthFrameReady += kinect_DepthFrameReady;
    //kinect.DepthStream.Range = DepthRange.Near;

    //kinect.ColorStream.Enable();
    //kinect.ColorFrameReady += kinect_ColorFrameReady;

}
```

```
private void kinect_DepthFrameReady
(object sender, DepthImageFrameReadyEventArgs e)
{

}
```

Caso você possua um Kinect na versão para Windows e desejar ativar o modo de proximidade você deve remover os comentários da linha de código que atribui o valor `Near` para a propriedade `Range` do fluxo de profundidade. Neste ponto, caso você inicialize a aplicação, irá perceber que a imagem de cores não está mais sendo exibida.

A primeira funcionalidade que vamos implementar será o reconhecimento de humanos pelo Kinect utilizando o *player segmentation data*. Vamos criar um método para fazer isso, mas antes é necessário também inicializar o **fluxo de esqueleto**, conforme mencionamos anteriormente.

Este método se chamará `ReconhecerHumanos`, ele receberá por parâmetro um quadro de profundidade (`DepthImageFrame`) e deverá retornar um objeto `BitmapSource` para preencher o componente de imagem de nossa aplicação. Para poder fazer a validação do pixel de acordo com o *player segmentation data*, devemos criar um array do tipo `DepthImagePixel`, que irá receber todas as informações do quadro de profundidade através do método `CopyDepthImagePixelDataTo`. Cada pixel deste array possui as propriedades `PlayerIndex`, `Depth` e `IsKnownDepth`, que representam respectivamente: o índice do usuário a qual o pixel pertence (zero quando o pixel não pertence a nenhum usuário), a distância do pixel ao sensor em milímetros e se o sensor consegue identificar a distância do pixel.

Após receber estas informações precisamos criar um novo array de bytes para gerar a imagem que será retornada no método. Este array deve ter quatro vezes o tamanho do array que representa os pixels de profundidade, pois conforme vimos antes, cada pixel de cor possui 4 bytes. Iremos percorrer este novo array e verificaremos se o byte referente ao mesmo pixel na imagem de profundidade pertence a um usuário ou não, caso pertença iremos pintar este pixel de verde. O código a seguir ilustra como esta implementação deve ser feita.

```
private BitmapSource ReconhecerHumanos
(DepthImageFrame quadro)
{
```

```
    if (quadro == null) return null;

    using (quadro)
    {
        DepthImagePixel[] imagemProfundidade =
        new DepthImagePixel[quadro.PixelDataLength];

        quadro.CopyDepthImagePixelDataTo(imagemProfundidade);

        byte[] bytesImagem = new byte[imagemProfundidade.Length * 4];

        for (int indice = 0; indice < bytesImagem.Length; indice+=4)
        {
            if (imagemProfundidade[indice / 4].PlayerIndex != 0)
            {
                bytesImagem[indice + 1] = 255;
            }
        }
        return BitmapSource.Create(quadro.Width, quadro.Height,
        96, 96, PixelFormats.Bgr32, null, bytesImagem,
        quadro.Width * 4);
    }
}
```

Você não deve ter muitos problemas em compreender o código do método `ReconhecerHumanos`, pois em diversos pontos ele é similar ao método `ObterImagemSensorRGB` criado no capítulo anterior. Note que nesta implementação inserimos o valor 255 apenas no byte referente à cor verde do RGB. Feito isso, você já deve conseguir executar a aplicação e obter um resultado semelhante ao da figura 5.2.
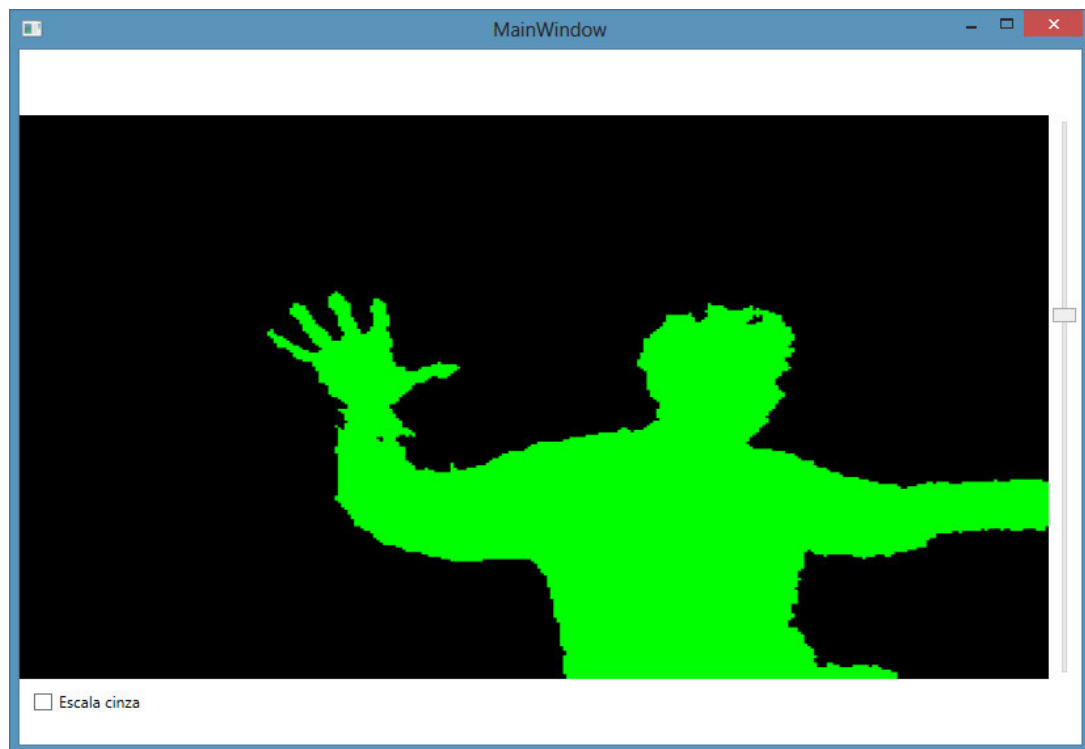
Figura 5.2: Reconhecedor de Humanos

Como você pode notar, esta funcionalidade não é perfeita, mas possui um resultado muito bom. Que tal agora fazermos uma combinação entre os dois sensores utilizados neste capítulo e no capítulo anterior? Pois bem, esta nova implementação fará com que o nosso componente de imagem da aplicação exiba todo o cenário através do sensor de cores (como fizemos no capítulo anterior), mas utilizará escala cinza somente nos pixels que estão em uma distância menor que 2 metros do sensor.

O primeiro passo para cumprir esta implementação é reativar o fluxo de cores. Agora nossos três fluxos estarão ligados simultaneamente, mas não iremos interpretar os eventos `ColorFrameReady` e `DepthFrameReady`. Ao invés disso utilizaremos o evento `AllFramesReady`, pois através dele poderemos acessar tanto o quadro de profundidade quanto o quadro de cores.

```
private void InicializarKinect(KinectSensor kinectSensor)
{
    kinect = kinectSensor;
```

```
    slider.Value = kinect.ElevationAngle;

    kinect.DepthStream.Enable();
    kinect.SkeletonStream.Enable();
    kinect.ColorStream.Enable();
    kinect.AllFramesReady += kinect_AllFramesReady;
}

private void kinect_AllFramesReady
(object sender, AllFramesReadyEventArgs e)
{

}
```

Precisamos alterar o método `ObterImagemSensorRGB` para retornar o array de bytes da imagem e não mais o objeto `BitmapSource`, pois agora precisamos reutilizar os bytes da imagem no método que reconhecerá a distância dos objetos — além disso, não é mais necessário deixar a lógica de programação referente à escala cinza neste método. Após estas alterações o método irá se parecer com o método ilustrado a seguir.

```
private byte[] ObterImagemSensorRGB(ColorImageFrame quadro)
{
    if (quadro == null) return null;

    using (quadro)
    {
        byte[] bytesImagem = new byte[quadro.PixelDataLength];
        quadro.CopyPixelDataTo(bytesImagem);

        return bytesImagem;
    }
}
```

O método ficou mais simples, pois agora ele apenas retorna os bytes da imagem. Agora, é a vez de criarmos um novo método chamado `ReconhecerDistância`, que deve receber por parâmetro os bytes da imagem e a distância máxima dos objetos para aplicar a escala cinza. A lógica para a escala cinza que foi removida do método anterior deve ser incorporada para este método dentro do bloco de validação da distância do sensor conforme código.

```
private void ReconhecerDistancia
(DepthImageFrame quadro, byte[] bytesImagem, int distanciaMaxima)
{
    if (quadro == null || bytesImagem == null) return null;

    using (quadro)
    {
        DepthImagePixel[] imagemProfundidade =
        new DepthImagePixel[quadro.PixelDataLength];

        quadro.CopyDepthImagePixelDataTo(imagemProfundidade);

        for (int indice = 0;
            indice < imagemProfundidade.Length;
            indice++)
        {
            if (imagemProfundidade[indice].Depth < distanciaMaxima)
            {
                int indiceImageCores = indice * 4;
                byte maiorValorCor =
                        Math.Max(bytesImagem[indiceImageCores],
                        Math.Max(bytesImagem[indiceImageCores + 1],
                        bytesImagem[indiceImageCores + 2]));

                bytesImagem[indiceImageCores] = maiorValorCor;
                bytesImagem[indiceImageCores + 1] = maiorValorCor;
                bytesImagem[indiceImageCores + 2] = maiorValorCor;
            }
        }
    }
```

Note que este método não retorna mais o objeto `BitmapSource`, isso ocorre pois criaremos a imagem dentro do método que interpreta o evento `AllFramesReady`. Neste método, sempre iremos captar a imagem do sensor de cores, mas aplicaremos o filtro de escala cinza criado no método `ReconhecerDistância` apenas quando o `CheckBox` "Escala cinza" estiver marcado, conforme o código a seguir.

```
private void kinect_AllFramesReady
(object sender, AllFramesReadyEventArgs e)
{
```

```
    byte[] imagem = ObterImagemSensorRGB(e.OpenColorImageFrame());

    if (chkEscalaCinza.IsChecked.HasValue &&
                chkEscalaCinza.IsChecked.Value)
        ReconhecerDistancia(e.OpenDepthImageFrame(), imagem, 2000);

    if (imagem != null)
        imagemCamera.Source =
        BitmapSource.Create(kinect.ColorStream.FrameWidth,
                            kinect.ColorStream.FrameHeight,
                            96, 96, PixelFormats.Bgr32, null,
                            imagem,
                            kinect.ColorStream.FrameBytesPerPixel
                            * kinect.ColorStream.FrameWidth);
}
```

Se tudo foi implementado de maneira correta, o resultado que teremos deverá ser similar ao resultado ilustrado pela figura 5.3.
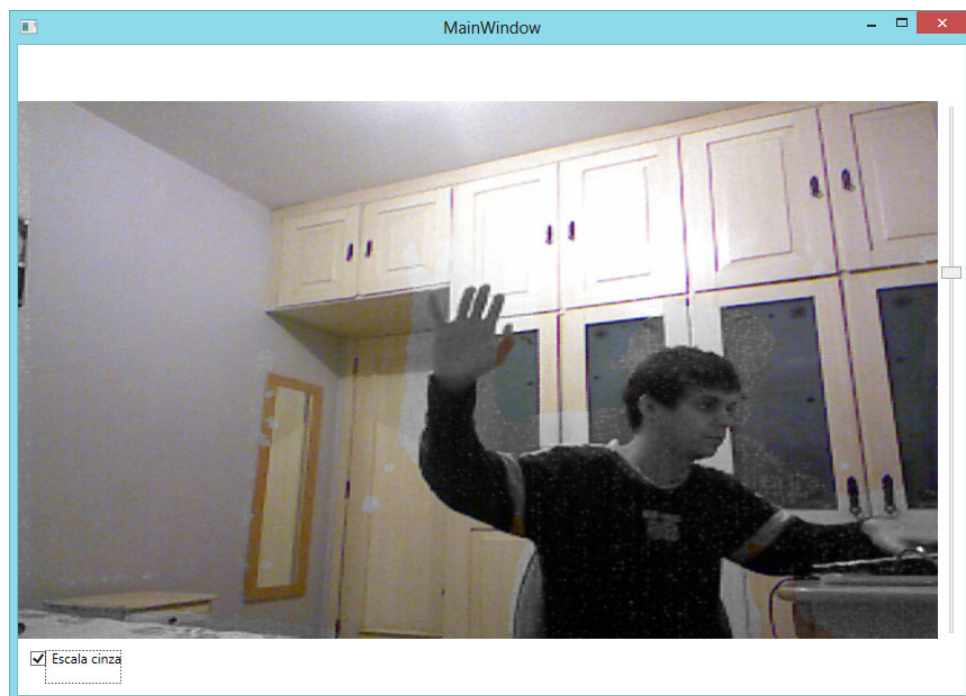
Figura 5.3: Escala cinza por distância

Você deve ter percebido que os pixels em escala cinza não estão perfeitamente alinhados onde deveriam, isso fica bastante claro na figura 5.3. Este problema de alinhamento ocorre devido ao fato de que os dois sensores: cor e profundidade ficam em locais físicos diferentes. Para nos ajudar com este problema, existem técnicas de mapeamento, que são feitas através da classe `CoordinateMapper`.

A classe `CoordinateMapper` faz a transformação de um sistema de coordenadas para outro entre os fluxos do Kinect. Infelizmente para fazer isso há uma perda no tamanho total da imagem, já que para equalizar os pontos é obrigatório que seja feita uma intersecção entre as visões dos diferentes sensores.

Utilizaremos a classe para mapear os pontos da imagem de profundidade em um array de tamanho 640x480 (devido ao formato dos fluxos selecionados) e verificar a profundidade através destes pontos mapeados. Além disso, uma outra técnica que nos ajuda a melhorar a qualidade do resultado é verificar se o ponto de profundidade é um ponto reconhecido pelo Kinect. Após estas mudanças o método `ReconhecerDistancia` deve ficar como o código adiante.

```
private void ReconhecerDistancia
(DepthImageFrame quadro, byte[] bytesImagem, int distanciaMaxima)
{
    if (quadro == null || bytesImagem == null) return;

    using (quadro)
    {
        DepthImagePixel[] imagemProfundidade =
        new DepthImagePixel[quadro.PixelDataLength];

        quadro.CopyDepthImagePixelDataTo(imagemProfundidade);

        DepthImagePoint[] pontosImagemProfundidade =
        new DepthImagePoint[640 * 480];

        kinect.CoordinateMapper
              .MapColorFrameToDepthFrame(kinect.ColorStream.Format,
              kinect.DepthStream.Format, imagemProfundidade,
              pontosImagemProfundidade);

        for (int i = 0; i < pontosImagemProfundidade.Length; i++)
        {
            var point = pontosImagemProfundidade[i];
            if ( point.Depth < distanciaMaxima &&
                 KinectSensor.IsKnownPoint(point) )
            {
                var pixelDataIndex = i * 4;

                byte maiorValorCor =
                        Math.Max(bytesImagem[pixelDataIndex],
                        Math.Max(bytesImagem[pixelDataIndex + 1],
                        bytesImagem[pixelDataIndex + 2]));

                bytesImagem[pixelDataIndex] = maiorValorCor;
                bytesImagem[pixelDataIndex + 1] = maiorValorCor;
                bytesImagem[pixelDataIndex + 2] = maiorValorCor;
            }
        }
    }
}
```

Neste ponto você deve executar a aplicação e obter um resultado similar ao da figura 5.4.
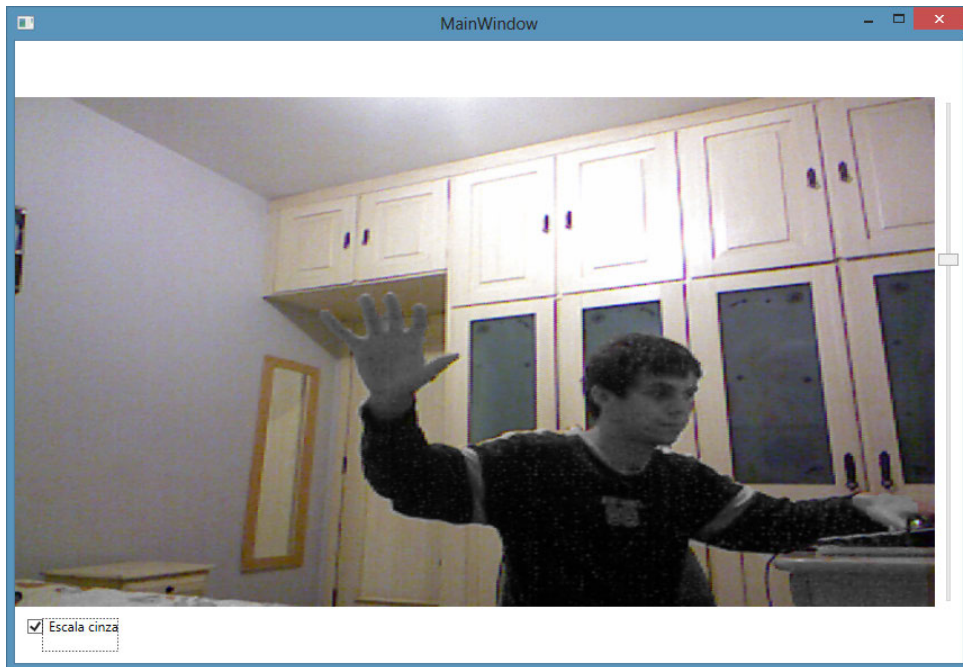
Figura 5.4: Escala cinza por distância (Melhorado)

No universo dos sensores praticamente nada é exato, infelizmente não podemos obter um resultado perfeito, mas como você deve ter notado após aplicar estas técnicas, o resultado se torna totalmente aceitável e com um nível de precisão bastante impressionante.

Antes de finalizarmos este capítulo é interessante ressaltar um teste nesta aplicação. Caso você aproxime demais sua mão do sensor ela voltará a ficar colorida, conforme figura 5.5, mesmo com a distância sendo menor do que a distância máxima para aplicar escala cinza. Você consegue imaginar por que isso ocorre?
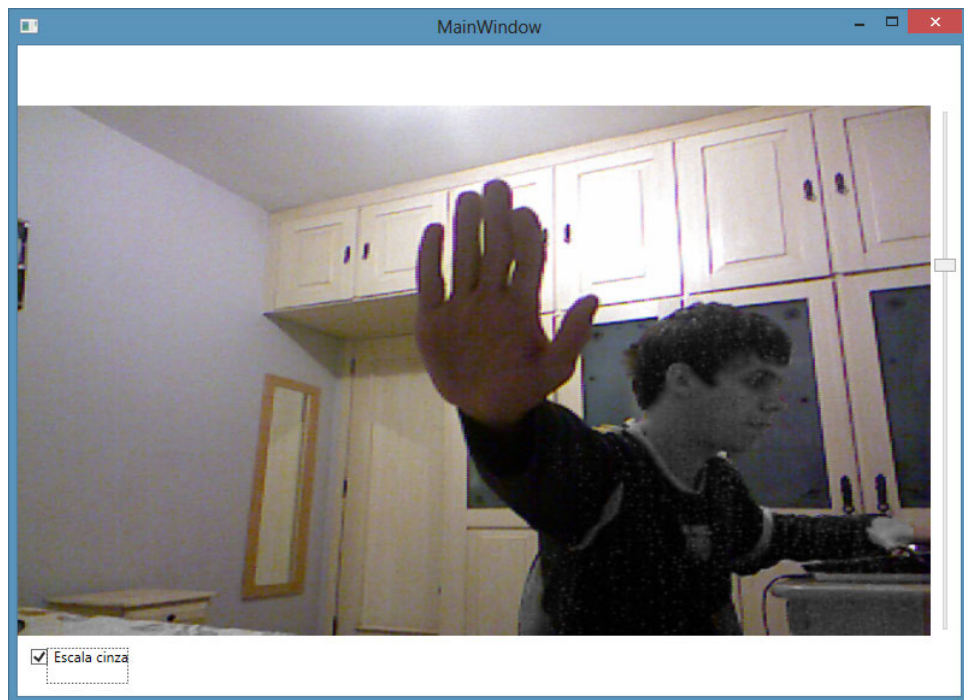
Figura 5.5: Escala cinza por distância (Teste)

Isso ocorre por conta de nossa verificação `KinectSensor.IsKnownPoint(point)`. No início deste capítulo foi mencionado que o Kinect possui uma distância mínima e uma máxima para conseguir identificar a profundidade de um ponto. Em nosso teste, quando a mão do usuário se aproximava mais do que a distância mínima do sensor ela passava a fazer parte dos pontos cuja distância o sensor não conhece.

Capítulo 6

Fluxo de Esqueleto do Usuário

O fluxo de esqueleto do usuário é um fluxo semelhante aos demais já apresentados nos capítulos anteriores, entretanto não há um sensor físico que o cria. Diferente dos outros, este fluxo é um conjunto de processamentos e de sensores que tornam o Kinect capaz de identificar usuários. O termo esqueleto de usuário refere-se à capacidade do Kinect que perceber o usuário e suas articulações, ou seja, com este fluxo conseguimos identificar por exemplo, as coordenadas X, Y e Z da mão do usuário.

Para efetuar o reconhecimento de diferentes tipos e tamanhos de esqueletos a Microsoft treinou uma rede neural artificial que processa as informações do ambiente tentando encontrar algum usuário. De acordo com a própria Microsoft sua rede neural foi treinada utilizando o conceito de Motion Capture (mocap) de diversos tipos.

Este fluxo é utilizado para criar aplicações baseadas em movimentos, sendo assim, ele pode ser considerado o fluxo mais importante do Kinect, pois praticamente toda aplicação utilizando o Kinect é feita baseada em movimentos.

6.1 Esqueleto do Usuário

As informações do usuário são disponibilizadas através de coordenadas X, Y e Z no espaço de coordenadas do Kinect. Como vimos no capítulo anterior, o sensor é capaz de reconhecer até 6 usuários simultaneamente, entretanto, apenas 2 destes usuários terão o esqueleto disponível para a aplicação.

O esqueleto do usuário conta com vinte articulações (frequentemente chamadas de Joints), sendo que cada uma possui suas próprias coordenadas. As articulações são pontos julgados importantes e através deles obtemos a descrição tridimensional do usuário. A figura 6.1 ilustra todas as articulações que podem ser obtidas através do sensor.
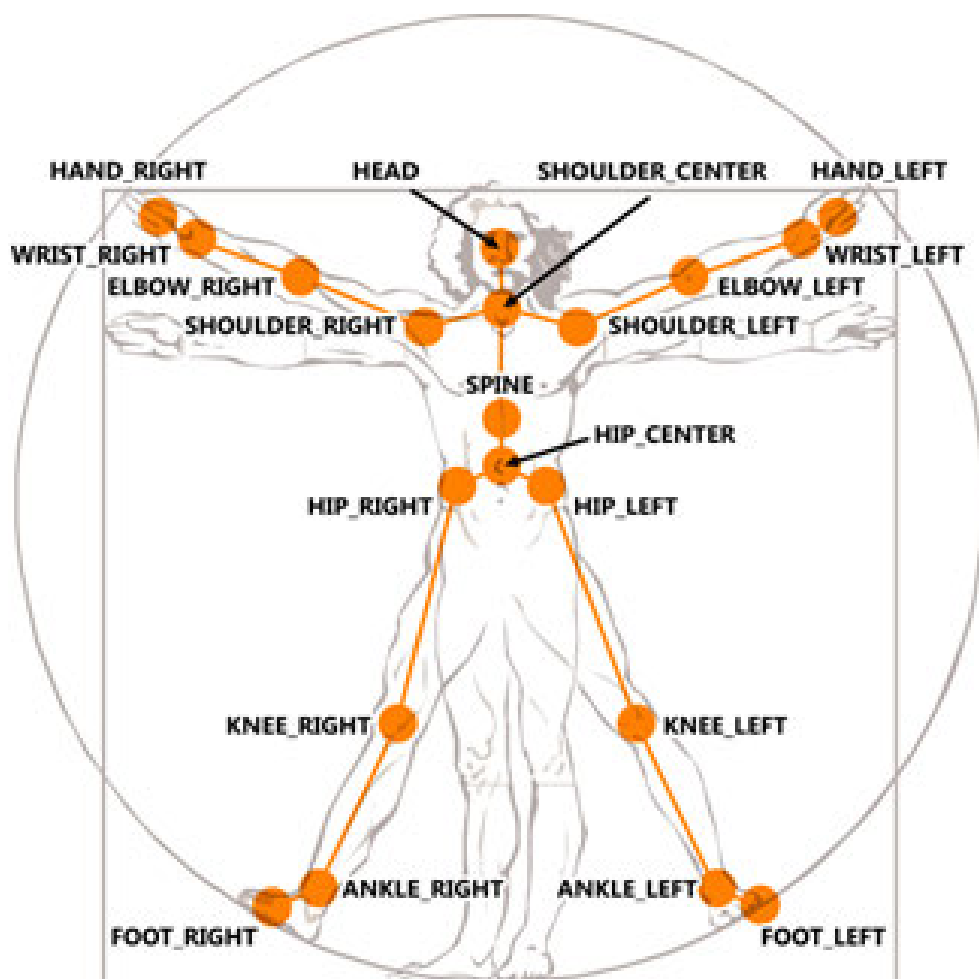
Figura 6.1: Articulações do usuário

Todas as articulações do Usuário estão interligadas — essas ligações no contexto de computação gráfica são chamadas de **ossos** ou **bones**, mesmo que a ligação não tenha um osso físico propriamente dito para ligá-las.

Movimentos

Existem dois tipos de movimentos que são utilizados a partir do esqueleto do usuário: **poses/posturas e gestos**.

Uma **pose** (também pode ser chamada de postura) é uma forma de manter o

corpo parado por determinado tempo até que isto tenha algum significado, por exemplo, quando um auxiliar de arbitragem no futebol nota que o atacante está em posição de impedimento ele ergue sua bandeira e se mantém assim até que o árbitro veja e aplique a regra. **Gestos** são os movimentos propriamente ditos, como por exemplo, um aceno de despedida.

Apesar de ser bastante simples obter informações a respeito do usuário, não se engane, uma aplicação baseada em movimentos foge bastante da trivialidade. Lembre-se que geralmente é necessário fazer com que os movimentos sejam detectados independente do esqueleto do usuário, o que pode se tornar um problema, já que as pessoas possuem tamanhos totalmente diferentes, tanto em tamanho total, quanto em tamanho de um *osso*. Além disso, quando falamos de gesto, há mais um complicador: o **tempo** de execução.

6.2 Desenhando o Esqueleto do Usuário

Vamos continuar com nossa aplicação dos capítulos anteriores. Agora acrescentaremos implementações utilizando o fluxo de esqueleto. Nossa implementação irá desenhar as articulações e o esqueleto do usuário sobre a imagem da câmera que já está implementada.

A primeira coisa a ser feita é desenhar o esqueleto do usuário. Não há um método nativo que faça este desenho para nós, então que tal criarmos um método de extensão para a classe `SkeletonFrame`? Faremos isso, mas dessa vez vamos fazer a implementação em outras classes em nossa própria aplicação e não em nossa DLL `AuxiliarKinect`. Isso porque o desenho será feito utilizando objetos WPF e não vamos acoplar nossa DLL auxiliar neste tipo de projeto.

Para começar vamos criar uma classe chamada `EsqueletoUsuarioAuxiliar`. Esta classe irá implementar os métodos para desenhar o esqueleto do usuário, além disso, ela deve receber um objeto do tipo `KinectSensor` em seu construtor, pois precisaremos dele para fazer o mapeamento do esqueleto, conforme o exemplo de código.

```
public class EsqueletoUsuarioAuxiliar
{
    private KinectSensor kinect;

    public EsqueletoUsuarioAuxiliar(KinectSensor kinect)
    {
```

```
        this.kinect = kinect;
    }

}
```

Agora precisamos criar dentro desta classe o método `ConverterCoordenadasArticulacao`. Ele irá retornar as coordenadas de uma articulação convertidas para um plano de duas dimensões. Utilizaremos novamente o `CoordinateMapper` para fazer esta conversão, mas dessa vez converteremos um `SkeletonPoint` para um `ColorImagePoint`. Este nosso novo método irá receber por parâmetro a articulação cujas coordenadas se deseja obter e dois valores do tipo `double` que devem conter os valores referentes à altura e largura do componente da janela em que iremos desenhar as articulações.

É importante lembrar-se de fazer uma regra de três para que a posição da articulação se enquadre no tamanho do componente que iremos desenhar nosso esqueleto, esta etapa pode ser ignorada caso o componente tenha exatamente o mesmo tamanho da imagem inserida no formato da `ColorStream` do Kinect, o método deve ficar similar ao código a seguir.

```
private ColorImagePoint ConverterCoordenadasArticulacao
(Joint articulacao, double larguraCanvas, double alturaCanvas)
{
    ColorImagePoint posicaoArticulacao =
    kinect.CoordinateMapper.MapSkeletonPointToColorPoint
    (articulacao.Position, kinect.ColorStream.Format);

    posicaoArticulacao.X = (int)
    (posicaoArticulacao.X * larguraCanvas) /
            kinect.ColorStream.FrameWidth;

    posicaoArticulacao.Y = (int)
    (posicaoArticulacao.Y * alturaCanvas) /
            kinect.ColorStream.FrameHeight;

    return posicaoArticulacao;
}
```

Com este método pronto já podemos converter a posição de nossas articulações, agora precisamos criar o componente que será desenhado na janela. Criaremos um método que se chama `CriarComponenteVisualArticulacao` para fazer este

trabalho. Este novo método também deve ser inserido na classe que estamos trabalhando. Para que seja possível desenhar as articulações em forma de círculos em um painel WPF iremos utilizar o objeto `Ellipse`, logo este método deve retornar um objeto deste tipo. Para uma maior personalização iremos fazer com que o método receba por parâmetro: o diâmetro do círculo, a largura da borda e a cor que o círculo será desenhado. A função deste método será utilizar estes parâmetros para a criação e configuração do `Ellipse`, conforme o código.

```
private Ellipse CriarComponenteVisualArticulacao
(int diametroArticulacao, int larguraDesenho, Brush corDesenho)
{
    Ellipse objetoArticulacao = new Ellipse();

    objetoArticulacao.Height = diametroArticulacao;
    objetoArticulacao.Width = diametroArticulacao;
    objetoArticulacao.StrokeThickness = larguraDesenho;
    objetoArticulacao.Stroke = corDesenho;
    return objetoArticulacao;
}
```

Já conseguimos obter as coordenadas da articulação e criar o componente visual para representá-la, agora criaremos um método que utilize estes dois criados anteriormente para fazer o desenho da articulação em sua posição. Chamaremos este novo método de `DesenharArticulacao`, ele deve receber por parâmetro um objeto `Joint` que é a representação da articulação do usuário e um objeto `Canvas` que é o painel WPF que utilizaremos para desenhar o esqueleto do usuário.

```
public void DesenharArticulacao
(Joint articulacao, Canvas canvasParaDesenhar)
{
    int diametroArticulacao =
    articulacao.JointType == JointType.Head ? 50 : 10;

    int larguraDesenho = 4;
    Brush corDesenho = Brushes.Red;

    Ellipse objetoArticulacao =
    CriarComponenteVisualArticulacao(
    diametroArticulacao, larguraDesenho,
    corDesenho);
```

```
    ColorImagePoint posicaoArticulacao =
    ConverterCoordenadasArticulacao(articulacao,
    canvasParaDesenhar.ActualWidth, canvasParaDesenhar.ActualHeight);

    double deslocamentoHorizontal =
    posicaoArticulacao.X - objetoArticulacao.Width / 2;

    double deslocamentoVertical =
    (posicaoArticulacao.Y - objetoArticulacao.Height / 2);

    if ( deslocamentoVertical >= 0 &&
         deslocamentoVertical < canvasParaDesenhar.ActualHeight &&
         deslocamentoHorizontal >= 0 &&
         deslocamentoHorizontal < canvasParaDesenhar.ActualWidth )
    {
        Canvas.SetLeft(objetoArticulacao, deslocamentoHorizontal);
        Canvas.SetTop(objetoArticulacao, deslocamentoVertical);
        Canvas.SetZIndex(objetoArticulacao, 100);

        canvasParaDesenhar.Children.Add(objetoArticulacao);
    }

}
```

Agora já temos um método para desenhar uma articulação do usuário, vamos fazer um teste? A primeira coisa a ser feita em nossa janela é substituir o componente `Image` por um componente do tipo `Canvas` e renomeá-lo para `canvasKinect`. Apenas com esta substituição já encontraremos um problema de compilação, pois o componente `Canvas` não possui a propriedade `Source`. No método que interpreta o evento `AllFramesReady` vamos substituir a linha que atribuía esta propriedade do componente `Image` por uma linha que altere a propriedade `Background` do componente `Canvas`. Também faremos com que o Canvas limpe os seus componentes cada vez que o método é chamado, para que o rastro do usuários seja apagado e redesenhado a cada quadro, conforme código a seguir.

```
private void kinect_AllFramesReady
(object sender, AllFramesReadyEventArgs e)
{
    byte[] imagem =
        ObterImagemSensorRGB(e.OpenColorImageFrame());
```

```
    if( chkEscalaCinza.IsChecked.HasValue &&
        chkEscalaCinza.IsChecked.Value)

        ReconhecerDistancia(e.OpenDepthImageFrame(),
                            imagem, 2000);

    if (imagem != null)
        canvasKinect.Background = new ImageBrush(
        BitmapSource.Create(kinect.ColorStream.FrameWidth,
                        kinect.ColorStream.FrameHeight,
                        96, 96, PixelFormats.Bgr32, null,
                        imagem,
                        kinect.ColorStream.FrameBytesPerPixel
                        * kinect.ColorStream.FrameWidth
                        )
        );

    canvasKinect.Children.Clear();
}
```

Após fazer isso iremos criar o método `DesenharEsqueletoUsuario` que receberá o `SkeletonFrame` por parâmetro e, assim como os outros métodos que utilizam o quadro de outros fluxos, também devemos utilizar o comando `using`. A classe `Skeleton` define a implementação virtual do esqueleto do usuário, então iremos utilizá-la para obter a articulação da mão do usuário. Isso pode ser feito através da propriedade `Joints`. Esta propriedade possui um indexador de acordo com o tipo da articulação conforme descrito na figura 6.1. Agora apenas para testes, iremos desenhar as elipses na posição das articulações das mãos do usuário, conforme o código.

```
private void DesenharEsqueletoUsuario
(SkeletonFrame quadro)
{
    if (quadro == null) return;

    using (quadro)
    {
        Skeleton[] esqueletos =
        new Skeleton[quadro.SkeletonArrayLength];
```

```
        quadro.CopySkeletonDataTo(esqueletos);

        IEnumerable<Skeleton> esqueletosRastreados =
         esqueletos.Where( esqueleto =>
         esqueleto.TrackingState ==
         SkeletonTrackingState.Tracked);

        if (esqueletosRastreados.Count() > 0)
        {
            Skeleton esqueleto =
             esqueletosRastreados.First();

            EsqueletoUsuarioAuxiliar funcoesEsqueletos =
             new EsqueletoUsuarioAuxiliar(kinect);

            funcoesEsqueletos.DesenharArticulacao(
             esqueleto.Joints[JointType.HandRight],
             canvasKinect );

            funcoesEsqueletos.DesenharArticulacao(
             esqueleto.Joints[JointType.HandLeft],
             canvasKinect );
        }
    }
}
```

Para finalizarmos precisamos fazer a chamada para o método acima, após a limpeza dos componentes do Canvas. Para obtermos o novo quadro do esqueleto que iremos enviar por parâmetro precisamos utilizar `e.OpenSkeletonFrame()` no método que interpreta o evento de todos os quadros. Com estas implementações prontas, já podemos obter resultados conforme a figura 6.2.
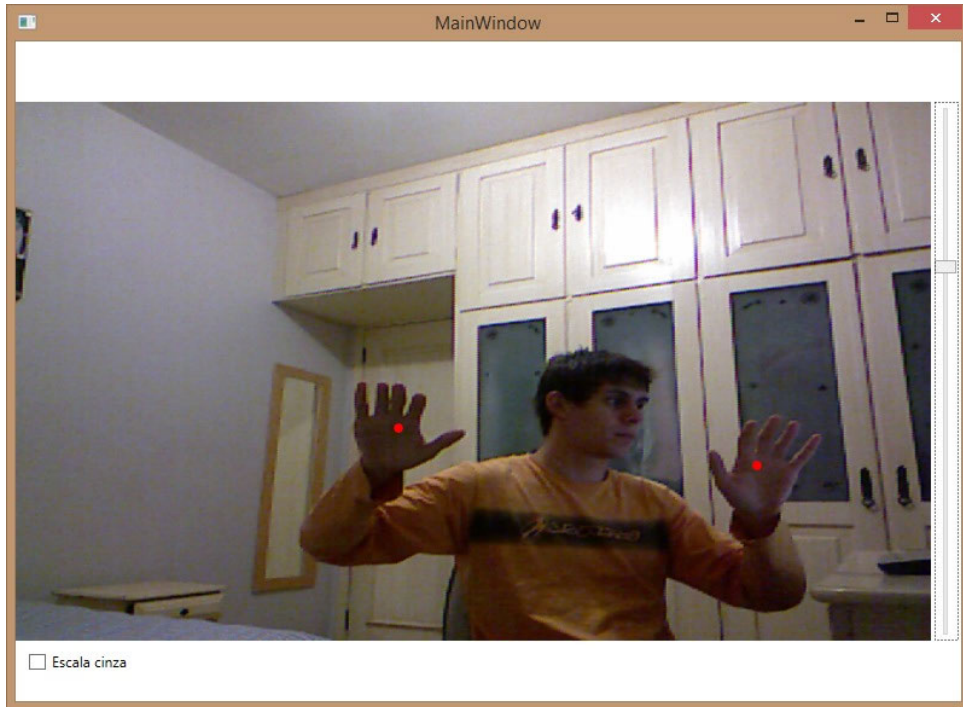
Figura 6.2: Desenhando a articulação das mãos

Neste ponto, já temos recursos para desenhar todas as articulações do usuário, mas iremos adicionar na classe `EsqueletoUsuarioAuxiliar` a funcionalidade para desenhar os *ossos* entre as articulações. Vamos fazê-la através de dois métodos: `DesenharOsso` e `CriarComponenteVisualOsso`. O método `CriarComponenteVisualOsso` deve retornar um objeto do tipo `Line` e receber por parâmetro a largura e cor da linha que iremos utilizar para desenhá-la e as coordenadas X e Y das articulações de destino e origem, conforme o exemplo.

```
private Line CriarComponenteVisualOsso
(int larguraDesenho, Brush corDesenho,
double origemX, double origemY, double destinoX, double destinoY)
{
    Line objetoOsso = new Line();

    objetoOsso.StrokeThickness = larguraDesenho;
    objetoOsso.Stroke = corDesenho;
```

```
    objetoOsso.X1 = origemX;
    objetoOsso.X2 = destinoX;

    objetoOsso.Y1 = origemY;
    objetoOsso.Y2 = destinoY;

    return objetoOsso;
}
```

Agora temos que criar o método `DesenharOsso`, ele utilizará duas articulações para criar um osso entre elas e inseri-lo no Canvas. A lógica deste método é bastante similar ao método `DesenharArticulacao`, pois teremos que converter as coordenadas 3D das duas articulações para um plano 2D utilizando o `CoordinateMapper` e usaremos o método descrito anteriormente para criar o objeto `Line` que será adicionado ao Canvas. O resultado deve ser similar ao código a seguir.

```
public void DesenharOsso
(Joint articulacaoOrigem, Joint articulacaoDestino,
        Canvas canvasParaDesenhar)
{
    int larguraDesenho = 4;
    Brush corDesenho = Brushes.Green;

    ColorImagePoint posicaoArticulacaoOrigem =
        ConverterCoordenadasArticulacao(
        articulacaoOrigem,
        canvasParaDesenhar.ActualWidth,
        canvasParaDesenhar.ActualHeight);

    ColorImagePoint posicaoArticulacaoDestino =
        ConverterCoordenadasArticulacao(
        articulacaoDestino,
        canvasParaDesenhar.ActualWidth,
        canvasParaDesenhar.ActualHeight);

    Line objetoOsso =
        CriarComponenteVisualOsso(larguraDesenho, corDesenho,
        posicaoArticulacaoOrigem.X, posicaoArticulacaoOrigem.Y,
        posicaoArticulacaoDestino.X, posicaoArticulacaoDestino.Y);
```

```
    if (Math.Max(objetoOsso.X1, objetoOsso.X2) <
        canvasParaDesenhar.ActualWidth &&
        Math.Min(objetoOsso.X1, objetoOsso.X2) > 0 &&
        Math.Max(objetoOsso.Y1, objetoOsso.Y2) <
        canvasParaDesenhar.ActualHeight &&
        Math.Min(objetoOsso.Y1, objetoOsso.Y2) > 0)

        canvasParaDesenhar.Children.Add(objetoOsso);
}
```

Agora sim, temos todos os métodos preparados. O próximo passo é criar uma nova classe no namespace Auxiliar chamada `Extensao`. Ela irá utilizar os métodos que preparamos acima para criar um método de extensão na classe `SkeletonFrame` para desenhar o esqueleto do usuário em um Canvas de forma coerente.

Métodos de Extensão

Na linguagem C# é possível criar métodos de extensão. Estes métodos são um tipo especial de métodos estáticos que permitem ao desenvolvedor adicionar funcionalidades em um tipo existente sem que você crie um tipo derivado, recompile ou modifique o código original. Este é um recurso poderoso e bastante utilizado nesta linguagem, não cabe ao escopo deste livro definir todas as possibilidades.

Para mais informações visite o link: http://msdn.microsoft.com/en-us/library/vstudio/bb383977.aspx

Para que a classe `Extensao` possa criar métodos de extensão para outros tipos é necessário que ela seja declarada como estática — para isso, utilize a palavra reservada `static`. Nesta classe iremos declarar o método estático `DesenharEsqueletoUsuario` para servir de extensão para a classe `SkeletonFrame`. Este método deve receber via parâmetros o sensor (`KinectSensor`) que ele utilizará para mapear o esqueleto e o canvas (`Canvas`) em que o esqueleto deve ser desenhado, além, é claro, do próprio objeto de extensão.

Para declarar o objeto que será estendido você deve incluí-lo como primeiro parâmetro antecedido pela palavra reservada `this`. Desta forma, este método será um método de extensão de uma instância da classe `SkeletonFrame`. Para garantir que

não haverá parâmetros inválidos, é necessário fazer uma validação nos parâmetros informados no método.

Caso todas as validações sejam cumpridas, iremos utilizar a lógica definida para obter o esqueleto do usuário no teste anterior e utilizar nossa classe `EsqueletoUsuarioAuxiliar` para primeiramente desenhar todas as articulações sem os ossos, seu método deve ficar similar ao código seguinte.

```
public static void DesenharEsqueletoUsuario
    ( this SkeletonFrame quadro,
      KinectSensor kinectSensor,
      Canvas canvasParaDesenhar )
{
    if (kinectSensor == null)
     throw new ArgumentNullException("kinectSensor");

    if (canvasParaDesenhar == null)
     throw new ArgumentNullException("canvasParaDesenhar");

    Skeleton[] esqueletos =
     new Skeleton[quadro.SkeletonArrayLength];

    quadro.CopySkeletonDataTo(esqueletos);

    IEnumerable<Skeleton> esqueletosRastreados =
        esqueletos.Where( esqueleto =>
        esqueleto.TrackingState ==
        SkeletonTrackingState.Tracked);

    if (esqueletosRastreados.Count() > 0)
    {
        Skeleton esqueleto =
         esqueletosRastreados.First();

        EsqueletoUsuarioAuxiliar esqueletoUsuarioAuxiliar =
         new EsqueletoUsuarioAuxiliar(kinectSensor);

        foreach (Joint articulacao in esqueleto.Joints)
            esqueletoUsuarioAuxiliar
             .DesenharArticulacao(articulacao, canvasParaDesenhar);
    }
```

Este método ficou bastante similar ao que já havíamos criado em nossa janela, mas além de passarmos a lógica para um método de extensão, podemos melhorar separando a lógica em partes. Obter o esqueleto do primeiro usuário é uma tarefa comum e bastante utilizada, então criaremos mais um método de extensão, que, dessa vez, retornará o esqueleto do primeiro usuário do sensor. Para fazer isso, basta utilizarmos a lógica já feita no método descrito anteriormente.

```
public static Skeleton ObterEsqueletoUsuario
 (this SkeletonFrame quadro)
{
    Skeleton esqueletoUsuario = null;
    Skeleton[] esqueletos =
     new Skeleton[quadro.SkeletonArrayLength];

    quadro.CopySkeletonDataTo(esqueletos);

    IEnumerable<Skeleton> esqueletosRastreados =
         esqueletos.Where( esqueleto =>
         esqueleto.TrackingState ==
         SkeletonTrackingState.Tracked);

    if (esqueletosRastreados.Count() > 0)
        esqueletoUsuario =
         esqueletosRastreados.First();

    return esqueletoUsuario;
}
```

Agora, iremos alterar o método `DesenharEsqueletoUsuario` para que ele reaproveite este método, de modo que, quando precisarmos somente do esqueleto do usuário sem desenhá-lo, podemos obtê-lo através do método deste método de extensão.

```
public static void DesenharEsqueletoUsuario
    ( this SkeletonFrame quadro,
      KinectSensor kinectSensor,
      Canvas canvasParaDesenhar )
{
    if (kinectSensor == null)
     throw new ArgumentNullException("kinectSensor");
```

```
    if (canvasParaDesenhar == null)
     throw new ArgumentNullException("canvasParaDesenhar");

    Skeleton esqueleto = ObterEsqueletoUsuario(quadro);
    if (esqueleto != null)
    {
        EsqueletoUsuarioAuxiliar esqueletoUsuarioAuxiliar =
         new EsqueletoUsuarioAuxiliar(kinectSensor);

        foreach (Joint articulacao in esqueleto.Joints)
            esqueletoUsuarioAuxiliar
             .DesenharArticulacao(articulacao, canvasParaDesenhar);
    }
```

Agora precisamos atualizar o método `DesenharEsqueletoUsuario` de nossa janela. Toda a lógica para obter o esqueleto do usuário já foi transferida para os métodos mostrados anteriormente, então não é mais necessário mantê-la no código da janela, basta que dentro do bloco `using` utilizemos o método de extensão, conforme o exemplo.

```
private void DesenharEsqueletoUsuario(SkeletonFrame quadro)
{
    if (quadro == null) return;

    using (quadro)
        quadro.DesenharEsqueletoUsuario(kinect, canvasKinect);
}
```

Note que retiramos toda a complexidade deste método e encapsulamos em um método de extensão, agora ficará muito mais fácil desenhar o esqueleto sempre que for necessário. Se você executar a aplicação, verá que todas as articulações já são exibidas e que a articulação da cabeça possui um diâmetro bem maior que as outras, conforme ilustra a figura 6.3
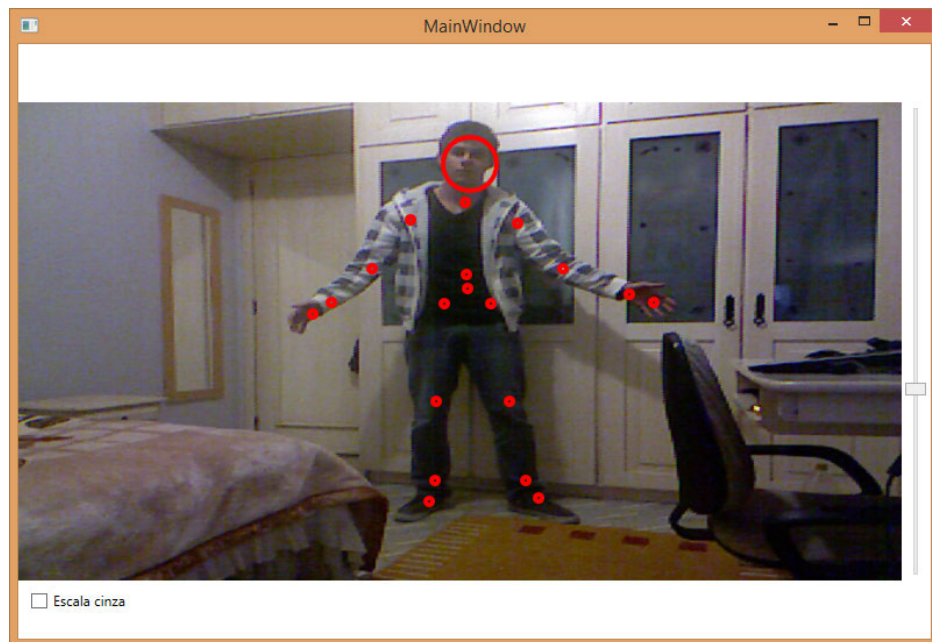
Figura 6.3: Desenhando todas as articulações

Como podemos notar na imagem acima já é possível ter uma boa noção de como ficará nosso resultado, agora para finalizarmos esta funcionalidade precisamos desenhar todos os ossos do esqueleto. Antes de fazermos isso, é preciso saber que há uma hierarquia coerente entre as articulações. Essa hierarquia define a relação entre as articulações, por exemplo, a articulação da cabeça está relacionada a articulação do centro dos ombros, que por sua vez está relacionada com o ombro direito e esquerdo e assim por diante. A figura 6.4 ilustra esta hierarquia.
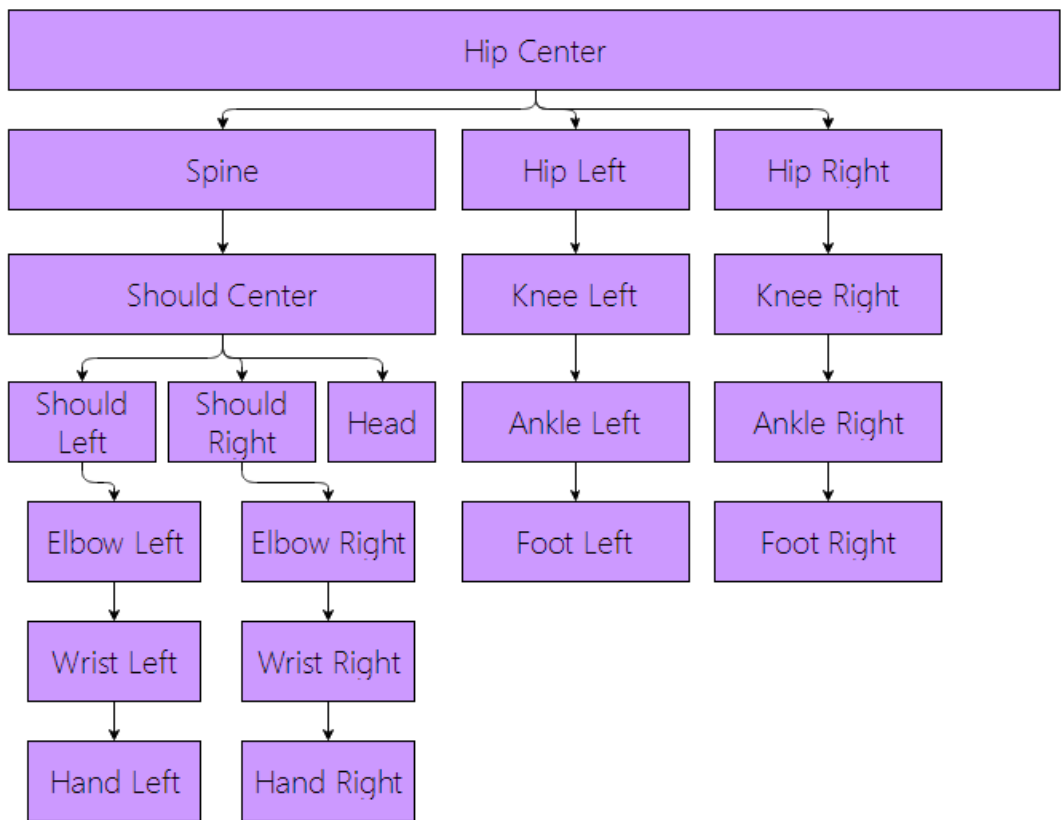
Figura 6.4: Hierarquia dos ossos

Através desta hierarquia também conseguimos obter informações sobre a relação das articulações, como por exemplo, a rotação de uma articulação em relação a outra. Para acessarmos estas informações devemos utilizar a propriedade `BoneOrientations` da classe `Skeleton`, esta propriedade é uma coleção de objetos do tipo `BoneOrientation`. Utilizaremos este objeto para obtermos a articulação inicial e final de cada osso de nosso esqueleto, então podemos alterar o laço de repetição do nosso método de extensão para percorrer a coleção de ossos. Dessa forma, teremos as informações sobre os ossos e sobre as articulações, fazendo o desenho completo do esqueleto do usuário, de acordo com o código abaixo.

```
public static void DesenharEsqueletoUsuario
    ( this SkeletonFrame quadro,
```

```
      KinectSensor kinectSensor,
      Canvas canvasParaDesenhar )
{
    if (kinectSensor == null)
     throw new ArgumentNullException("kinectSensor");

    if (canvasParaDesenhar == null)
     throw new ArgumentNullException("canvasParaDesenhar");

    Skeleton esqueleto = ObterEsqueletoUsuario(quadro);
    if (esqueleto != null)
    {
        EsqueletoUsuarioAuxiliar esqueletoUsuarioAuxiliar =
         new EsqueletoUsuarioAuxiliar(kinectSensor);

        foreach (BoneOrientation osso in esqueleto.BoneOrientations)
        {
            esqueletoUsuarioAuxiliar
             .DesenharOsso(esqueleto.Joints[osso.StartJoint],
                           esqueleto.Joints[osso.EndJoint],
                           canvasParaDesenhar);

            esqueletoUsuarioAuxiliar
             .DesenharArticulacao
             (esqueleto.Joints[osso.EndJoint], canvasParaDesenhar);
        }
    }
```

Enfim temos o resultado esperado. O esqueleto já pode ser desenhado totalmente com uma simples chamada ao método de extensão, o resultado é ilustrado pela figura 6.5.
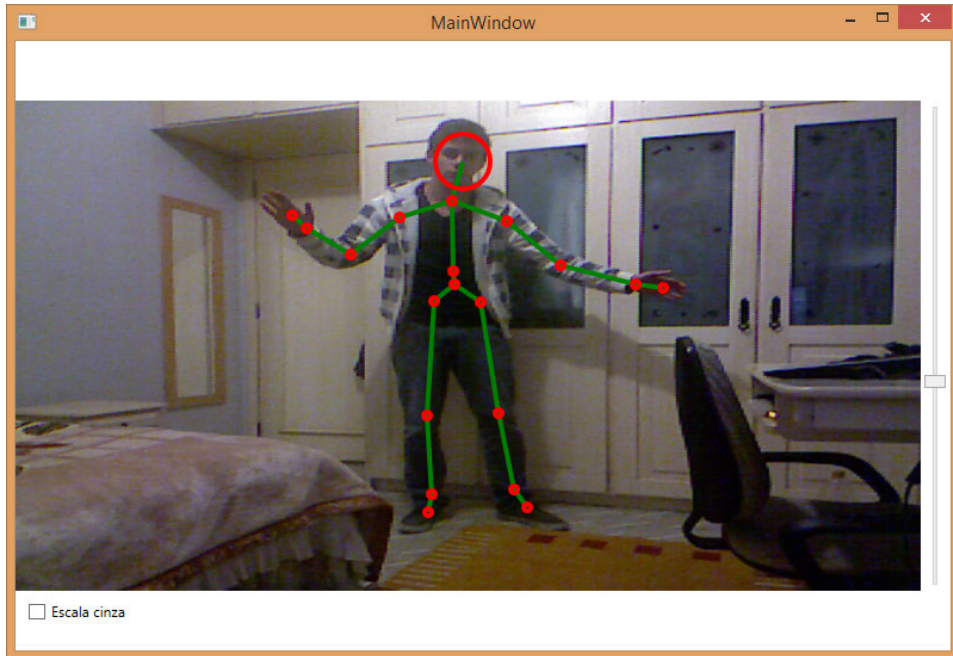
Figura 6.5: Desenhando o esqueleto do usuário

Agora vamos criar um novo `CheckBox` para identificar quando o esqueleto do usuário deve ser desenhado. Em nossa janela iremos agrupar o `CheckBox` já existente para aplicar o filtro de escala cinza em um `StackPanel`. O trecho de XAML de sua janela deve ficar similar a este:

```
<StackPanel Grid.Row="2" Orientation="Horizontal">
    <CheckBox Name="chkEscalaCinza"
     Content="Escala cinza" Margin="10,10,0,10"
     HorizontalAlignment="Left" />

    <CheckBox Name="chkEsqueleto"
     Content="Esqueleto do Usuário"
     HorizontalAlignment="Left"
     Margin="10,10,0,10"/>

</StackPanel>
```

Além de inserir o componente na janela, também é necessário fazer a validação

deste `CheckBox` antes de utilizar o método para desenhar o esqueleto, então nosso método que interpreta o evento `AllFramesReady` deve ficar assim:

```
private void kinect_AllFramesReady
(object sender, AllFramesReadyEventArgs e)
{
    byte[] imagem = ObterImagemSensorRGB(e.OpenColorImageFrame());

    if( chkEscalaCinza.IsChecked.HasValue &&
        chkEscalaCinza.IsChecked.Value )
        ReconhecerDistancia(e.OpenDepthImageFrame(),
                            imagem, 2000);

    if (imagem != null)
        canvasKinect.Background = new ImageBrush(
        BitmapSource
         .Create( kinect.ColorStream.FrameWidth,
                  kinect.ColorStream.FrameHeight,
                  96, 96, PixelFormats.Bgr32, null,
                  imagem,
                  kinect.ColorStream.FrameBytesPerPixel
                  * kinect.ColorStream.FrameWidth
                )
        );

    canvasKinect.Children.Clear();

    if ( chkEsqueleto.IsChecked.HasValue &&
         chkEsqueleto.IsChecked.Value )
        DesenharEsqueletoUsuario(e.OpenSkeletonFrame());
}
```

Feito isso, por padrão visualizaremos apenas a imagem da câmera do Kinect e temos a opção de marcar o `CheckBox` "Esqueleto do Usuário" para quando desejarmos visualizar o esqueleto do usuário.

Capítulo 7

Rastreando e Identificando Movimentos

Neste capítulo continuaremos a utilizar o fluxo de esqueleto como no capítulo anterior, mas dessa vez focaremos na interação com o aplicativo através de movimentos. Este método de interação pode trazer uma grande experiência ao usuário, porém temos de levar em consideração que há alguns desafios para implementar uma boa usabilidade com nossos movimentos. É importante para o usuário que os movimentos tenham dificuldades de acordo com seu contexto, por exemplo, em um jogo pode ser que um movimento complicado faça parte do desafio, porém se o usuário está utilizando movimentos para navegar em um aplicativo ou acionar atalhos, eles devem ser fáceis e naturais.

Os movimentos precisam ser bem diferenciados, evite criar movimentos similares com funcionalidades diferentes, pois a interpretação de movimentos está sujeita a erros e é frustrante para a experiência do usuário acionar uma funcionalidade por acaso ao tentar executar outra.

É importante conhecer o público alvo de sua aplicação e é necessário tomar precauções quanto ao tamanho do esqueleto de cada usuário e o tamanho do osso entre as articulações de cada usuário. Procure sempre fazer comparações proporcionais ao próprio esqueleto, dessa forma a interpretação do gesto se mantém consistente independente do tamanho do esqueleto.

Testar e experimentar é fundamental quando se fala de aplicações voltadas a interfaces naturais. Você pode desenvolver uma aplicação baseada em movimentos que funciona perfeitamente para você, mas que não atende seu público alvo, portanto procure fazer testes, avaliar e reimplementar quantas vezes forem necessário a fim de melhorar cada vez mais a experiência do usuário.

A ideia principal da implementação do capítulo anterior é que o usuário visualize seu esqueleto apenas quando desejar, isso é uma boa ideia, mas há um problema de usabilidade em nossa implementação. Quando o usuário estiver se vendo no vídeo exibido por nossa aplicação, ele poderá estar longe de seu computador e isso irá impossibilitá-lo de visualizar seu esqueleto naquele momento. Este mesmo problema também pode ser aplicado para quando o usuário deseja aplicar o filtro de escala cinza.

Para resolvermos isso iremos implementar o rastreamento e a identificação de movimentos para acionar atalhos, o que pode ser bastante trabalhoso, mas montaremos uma estrutura na DLL `AuxiliarKinect` para nos ajudar.

7.1 Iniciando a Estrutura Base para Detectar Movimentos

Primeiro vamos criar uma nova pasta no projeto `AuxiliarKinect` chamada `Movimentos`. Nela serão criadas todas as classes ligadas à lógica para rastrear e identificar as poses e gestos que desenvolveremos ao longo deste livro.

Todos os movimentos em nossa aplicação poderão estar em um dos três estados de rastreamento: não identificado, em execução e identificado. Para podermos ter esses valores em nossa aplicação iremos criar um arquivo com a enumeração `EstadoRastreamento`, conforme o código a seguir.

```
public enum EstadoRastreamento
{
    NaoIdentificado,
    EmExecucao,
```

```
    Identificado
}
```

Após isso, criaremos a primeira classe que se chamará `Movimento`, ela será uma classe base para todos os movimentos (poses e gestos) de nossa aplicação. Esta classe deve ser abstrata e possuir as propriedades e métodos que todo tipo de movimento tem. Todo e qualquer tipo de movimento envolve tempo de execução, sendo assim, é necessário que a classe `Movimento` tenha um contador para quadros em que o movimento está em execução. Além disso, ele deve conter um nome e deve possuir métodos que permitam validar se a posição do usuário está de acordo com o movimento que ele está tentando executar. A classe deve ficar similar à abaixo:

```
public abstract class Movimento
{
    protected int ContadorQuadros { get; set; }
    public string Nome { get; set; }

    public abstract EstadoRastreamento Rastrear(
        Skeleton esqueletoUsuario);

    protected abstract bool PosicaoValida(
        Skeleton esqueletoUsuario);
}
```

Para que uma pose tenha validade é necessário que ela esteja sendo feita por um período de tempo. A etapa que avalia a pose ao longo do tempo chama-se **rastreamento** e a etapa que ocorre quando a pose é reconhecida como válida chama-se **identificação**.

Agora criaremos a classe abstrata que deverá ser utilizada como base para todas as poses de nossa aplicação. Ela deverá ser chamada de `Pose` e deve herdar a classe `Movimento`, bem como possuir a propriedade que marca em qual frame a pose é reconhecida e a implementação do método `Rastrear` herdado pela classe `Movimento`. Mas por enquanto deixaremos o método `Rastrear` em branco e voltaremos aqui mais tarde.

```
public abstract class Pose : Movimento
{
    protected int QuadroIdentificacao { get; set; }

    public override EstadoRastreamento Rastrear(
```

```
        Skeleton esqueletoUsuario)
    {

    }
}
```

Além das classes bases para os movimentos também precisamos de uma classe que irá rastrear e identificar as poses, então no namespace `Movimentos` criaremos a interface `IRastreador` e a classe `Rastreador`.

A interface `IRastreador` deve ser implementada por todos os rastreadores. Em seu corpo deve ser definido o método `Rastrear`, que recebe por parâmetro o esqueleto do usuário e não precisa retornar nenhum tipo de valor, e a propriedade `EstadoAtual`, que deve ser somente leitura.

```
public interface IRastreador
{
    void Rastrear(Skeleton esqueletoUsuario);
    EstadoRastreamento EstadoAtual { get; }
}
```

A classe `Rastreador` irá fazer o rastreamento de um determinado movimento, seja ele pose ou gesto, implementando o método `Rastrear` da interface criada anteriormente. Para configurarmos que tipo de pose ou gesto iremos rastrear é necessária a utilização de **generics**.

Generics

Generics é uma funcionalidade da linguagem C# que permite ao desenvolvedor criar um design de classes que podem aplicar funcionalidades a um determinado tipo que será definido apenas na aplicação final. Classes genéricas maximizam bastante a reutilização de código, um exemplo de utilização é a coleção List<T>, onde T pode ser um tipo qualquer. Dessa forma, é possível criar listas de qualquer tipo de objeto e ela terá os mesmos métodos. Para mais informações acesse:

http://msdn.microsoft.com/en-us/library/512aeb7t

Apesar de usar generics, nossa classe `Rastreador` deve rastrear apenas objetos que herdem da classe `Movimento` e que sejam objetos concretos. Podemos definir

isso na própria declaração da classe utilizando a palavra reservada `where` e definindo as condições desejadas, conforme o código.

```
public class Rastreador<T> : IRastreador where T : Movimento, new()
{
}
```

Com este cabeçalho definimos que a classe `Rastreador` pode ser utilizada com um tipo genérico, que ela implementa a interface `IRastreador` e que ela permite somente tipos que herdem a classe `Movimento` e que possuem um construtor. Internamente, nosso construtor terá de ter um atributo privado para controlar o próprio objeto referente ao movimento que está sendo rastreado. Além disso, precisaremos criar um evento para quando o movimento é reconhecido e outro para notificar o progresso do rastreamento. No construtor padrão desta classe iremos inicializar o objeto que armazena o movimento que será rastreado.

```
public class Rastreador<T> where T : Movimento, new()
{
    private T movimento;
    public EstadoRastreamento EstadoAtual { get; private set; }
    public event EventHandler MovimentoIdentificado;
    public event EventHandler MovimentoEmProgresso;

    public Rastreador()
    {
        EstadoAtual = EstadoRastreamento.NaoIdentificado;
        movimento = Activator.CreateInstance<T>();
    }
}
```

A classe `Activator` é utilizada para invocar o construtor do tipo definido como genérico de nossa classe (existem formas mais performáticas, mas por simplicidade utilizaremos esta no exemplo). Para finalizar a implementação de nosso rastreador iremos implementar o método `Rastrear` da interface e nele iremos verificar através do método `Rastrear` da classe `Movimento` se o mesmo foi identificado ou está em progresso. De acordo com o estado do rastreamento, chamaremos os eventos. Não esqueça que é necessário verificar se o evento está nulo antes de chamá-lo. O código final desta classe deve ficar semelhante ao código a seguir.

```
public class Rastreador<T> : IRastreador where T : Movimento, new()
{
```

```
    private T movimento;
    public EstadoRastreamento EstadoAtual { get; private set; }
    public event EventHandler MovimentoIdentificado;
    public event EventHandler MovimentoEmProgresso;

    public Rastreador()
    {
        EstadoAtual = EstadoRastreamento.NaoIdentificado;
        movimento = Activator.CreateInstance<T>();
    }

    public void Rastrear(Skeleton esqueletoUsuario)
    {
        EstadoRastreamento novoEstado =
            movimento.Rastrear(esqueletoUsuario);

        if (novoEstado == EstadoRastreamento.Identificado &&
            EstadoAtual != EstadoRastreamento.Identificado)
            ChamarEvento(MovimentoIdentificado);

        if (novoEstado == EstadoRastreamento.EmExecucao &&
            (EstadoAtual == EstadoRastreamento.EmExecucao ||
              EstadoAtual == EstadoRastreamento.NaoIdentificado))
            ChamarEvento(MovimentoEmProgresso);

        EstadoAtual = novoEstado;
    }

    private void ChamarEvento(EventHandler evento)
    {
        if (evento != null)
            evento(movimento, new EventArgs());
    }

}
```

Com isso, nossa estrutura básica para poses já está pronta, voltaremos mais tarde para implementar a parte de gestos de nossa estrutura, agora vamos implementar as poses que de fato serão rastreadas.

7.2 Utilizando Poses em Nossa Aplicação

A primeira pose que iremos criar é uma pose simples, conhecida como "T". Esta pose consiste em manter os dois braços esticados para os lados em altura paralela aos ombros, fazendo com que o formato do esqueleto do usuário se assemelhe com a letra "T" (motivo do nome da pose).

Agora iremos criar uma nova pasta chamada `Poses`, que deve ficar dentro da pasta `Movimentos` e nela devem ser inseridas todas as classes referentes a poses interpretadas por nossa aplicação. Nela iremos incluir a classe `PoseT`, lembre-se de que a classe `PoseT` deve ser uma classe concreta e herdar a classe `Pose`.

No construtor da classe `PoseT` já podemos inserir os valores nas propriedades `Nome` e `QuadroIdentificacao` — neste exemplo utilize respectivamente os valores "PoseT" e 10. Quando você incluiu a herança da classe `Pose` você tornou necessário que a classe `PoseT` implemente a sobrescrita do método `PosicalValida` (não é necessário sobrescrever o método `Rastrear` pois este já foi sobrescrito na classe `Pose`). Nesse momento sua classe deve estar semelhante à classe a seguir.

```
public class PoseT : Pose
{
    public PoseT()
    {
        this.Nome = "PoseT";
        this.QuadroIdentificacao = 10;
    }

    protected override bool PosicaoValida(Skeleton esqueletoUsuario)
    {

    }
}
```

Precisamos agora implementar estes métodos, mas antes de começarmos é importante ter em mente que as comparações entre as articulações não devem ser feitas de forma exata, ou seja, é inviável exigir ao usuário que as posições sejam exatamente iguais. Para resolver isso devemos utilizar uma margem de erro na comparação dos valores entre quaisquer posições. Antes de implementar o método `PosicaoValida` iremos criar uma nova classe estática chamada `Util` no namespace `FuncoesBasicas`. A classe `Util` será utilizada para fazer operações comuns, como por exemplo, a comparação utilizando margem de erro.

Após criar a classe `Util` criaremos o método `CompararComMargemErro` que deve receber por parâmetro três valores do tipo `double`, um valor para ser utilizado como margem de erro e os dois valores que serão comparados. Segue a implementação desta classe.

```
public static class Util
{
    public static bool CompararComMargemErro
     (double margemErro, double valor1, double valor2)
    {
        return valor1 >= valor2 - margemErro &&
               valor1 <= valor2 + margemErro;
    }
}
```

Podemos voltar agora para nossa classe `PoseT` e finalmente implementar o método `PosicaoValida`. Este método deve utilizar as articulações do esqueleto do usuário que é recebido por parâmetro para validar se o usuário está na posição relativa à pose em questão, neste caso, a pose "T".

Na validação desta pose devemos comparar a altura(eixo Y) e a distância(eixo Z) das mãos e dos cotovelos em relação ao centro dos ombros e se o eixo X das mãos estão mais distantes de seu respectivo cotovelo. Todas essas comparações que envolvem igualdade devem utilizar margem de erro — neste caso utilizaremos como margem de erro o valor 0.30, mas você pode ajustá-lo conforme sua necessidade de precisão, conforme o código.

```
protected override bool PosicaoValida
 (Skeleton esqueletoUsuario)
{
    Joint centroOmbros =
     esqueletoUsuario.Joints[JointType.ShoulderCenter];

    Joint maoDireita =
     esqueletoUsuario.Joints[JointType.HandRight];

    Joint cotoveloDireito =
     esqueletoUsuario.Joints[JointType.ElbowRight];

    Joint maoEsquerda =
     esqueletoUsuario.Joints[JointType.HandLeft];
```

```
Joint cotoveloEsquerdo =
 esqueletoUsuario.Joints[JointType.ElbowLeft];

double margemErro = 0.30;

bool maoDireitaAlturaCorreta =
 Util.CompararComMargemErro(margemErro,
 maoDireita.Position.Y, centroOmbros.Position.Y);

bool maoDireitaDistanciaCorreta =
 Util.CompararComMargemErro(margemErro,
 maoDireita.Position.Z, centroOmbros.Position.Z);

bool maoDireitaAposCotovelo =
 maoDireita.Position.X > cotoveloDireito.Position.X;

bool cotoveloDireitoAlturaCorreta =
 Util.CompararComMargemErro(margemErro,
 cotoveloDireito.Position.Y, centroOmbros.Position.Y);

bool cotoveloEsquerdoAlturaCorreta =
 Util.CompararComMargemErro(margemErro,
cotoveloEsquerdo.Position.Y, centroOmbros.Position.Y);

bool maoEsquerdaAlturaCorreta =
 Util.CompararComMargemErro(margemErro,
 maoEsquerda.Position.Y, centroOmbros.Position.Y);

bool maoEsquerdaDistanciaCorreta =
 Util.CompararComMargemErro(margemErro,
 maoEsquerda.Position.Z, centroOmbros.Position.Z);

bool maoEsquerdaAposCotovelo =
 maoEsquerda.Position.X < cotoveloEsquerdo.Position.X;

return maoDireitaAlturaCorreta &&
       maoDireitaDistanciaCorreta &&
       maoDireitaAposCotovelo &&
       cotoveloDireitoAlturaCorreta &&
       maoEsquerdaAlturaCorreta &&
```

```
            maoEsquerdaDistanciaCorreta &&
            maoEsquerdaAposCotovelo &&
            cotoveloEsquerdoAlturaCorreta;
}
```

Com este método implementado já conseguimos validar se nosso usuário está ou não na posição correta em relação à pose "T", mas ainda precisamos implementar o método `Rastrear` na classe `Pose`. Este método deve verificar se o usuário está na pose correta de acordo com o método `PosicaoValida` e retornar o estado da pose de acordo com o número de quadros que o usuário está na posição correta.

A lógica para identificação de todas as poses é igual, por este motivo optou-se por implementar o método `Rastrear` na classe base de todas as poses. É necessário verificar se o esqueleto do usuário ainda está sendo rastreado e se a posição em que ele está é válida para a pose que estamos rastreando. Como já mencionado isso pode ser feito através de nosso método `PosicaoValida`. Caso o usuário esteja em uma posição válida, é necessário validar também o número de quadros em que ele está nesta posição; se o número de quadros já atingiu o número definido na propriedade `QuadroIdentificacao`, o estado da classe deve ser alterado para `Identificado`; caso ainda não tenha atingido, o estado da classe deve ser alterado para `EmExecucao` e o contador deve ser incrementado. Se a posição do usuário não for válida na primeira condição o estado da pose deve ser `NaoIdentificado` e o contador de quadros deve zerar.

```
public override EstadoRastreamento Rastrear
 (Skeleton esqueletoUsuario)
{
    EstadoRastreamento novoEstado;
    if ( esqueletoUsuario != null &&
         PosicaoValida(esqueletoUsuario) )
    {
        if (QuadroIdentificacao == ContadorQuadros)
            novoEstado = EstadoRastreamento.Identificado;
        else
        {
            novoEstado = EstadoRastreamento.EmExecucao;
            ContadorQuadros += 1;
        }
    }
    else
```

```
    {
        novoEstado = EstadoRastreamento.NaoIdentificado;
        ContadorQuadros = 0;
    }
    return novoEstado;
}
```

Por fim, criaremos uma propriedade que contenha o percentual de progresso de nossa pose de acordo com as propriedades `ContadorQuadros` e `QuadroIdentificacao`, conforme o código.

```
public int PercentualProgresso
{
    get
    {
        return ContadorQuadros * 100 / QuadroIdentificacao;
    }
}
```

Agora já podemos compilar este projeto e retornar para nossa janela para fazer uso deste código!

Da mesma forma que criamos um atributo na janela para controlar o kinect em todos os métodos faremos isso criando um objeto do tipo `List<IRastreador>`, que conterá todos os rastreadores de movimentos que serão executados. Iremos criar um método em nossa janela chamado `InicializarRastreadores`. Ele deve inicializar os rastreadores de nossa janela (por enquanto só haverá um) e definir o método que será chamado no disparo do evento `MovimentoIdentificado` e `MovimentoEmProgresso`, quando houver. Neste caso, o método que será disparado quando o movimento for identificado apenas deve alterar a propriedade `IsChecked` de nosso `CheckBox` relacionado ao esqueleto do usuário. A implementação dessas funcionalidades deve seguir conforme o seguinte código.

```
private void InicializarRastreadores()
{
    rastreadores = new List<IRastreador>();

    Rastreador<PoseT> rastreadorPoseT = new Rastreador<PoseT>();
    rastreadorPoseT.MovimentoIdentificado += PoseTIdentificada;

    rastreadores.Add(rastreadorPoseT);
```

```
}

private void PoseTIdentificada(object sender, EventArgs e)
{
    chkEsqueleto.IsChecked = !chkEsqueleto.IsChecked;
}
```

Note que, quando criamos um objeto do tipo `Rastreador`, já definimos qual o movimento que ele irá rastrear através do uso de generics. Apesar do trecho de código acima não ilustrar, é necessário que você defina o objeto `rastreadores` no escopo da classe e que chame o método `InicializarRastreadores` no construtor da janela.

Agora você deve renomear o método `DesenharEsqueletoUsuario` para `FuncoesEsqueletoUsuario`, pois ele também irá fazer o rastreamento dos movimentos definidos em nossa lista. Dentro do bloco `using` deste método devemos chamar o método de extensão para obtermos o primeiro esqueleto e percorrer a lista rastreadores passando-o por parâmetro ao método `Rastrear`. Após isso já poderemos testar o rastreamento de nossos movimentos.

```
private void DesenharEsqueletoUsuario(SkeletonFrame quadro)
{
    if (quadro == null) return;

    using (quadro)
    {
        Skeleton esqueletoUsuario =
         quadro.ObterEsqueletoUsuario();

        foreach (IRastreador rastreador in rastreadores)
            rastreador.Rastrear(esqueletoUsuario);

        if ( chkEsqueleto.IsChecked.HasValue &&
             chkEsqueleto.IsChecked.Value )
            quadro.DesenharEsqueletoUsuario(kinect, canvasKinect);
    }
}
```

Agora você já deve conseguir utilizar a pose "T" para ativar ou desativar a visualização do esqueleto do usuário, conforme ilustra a figura 7.1
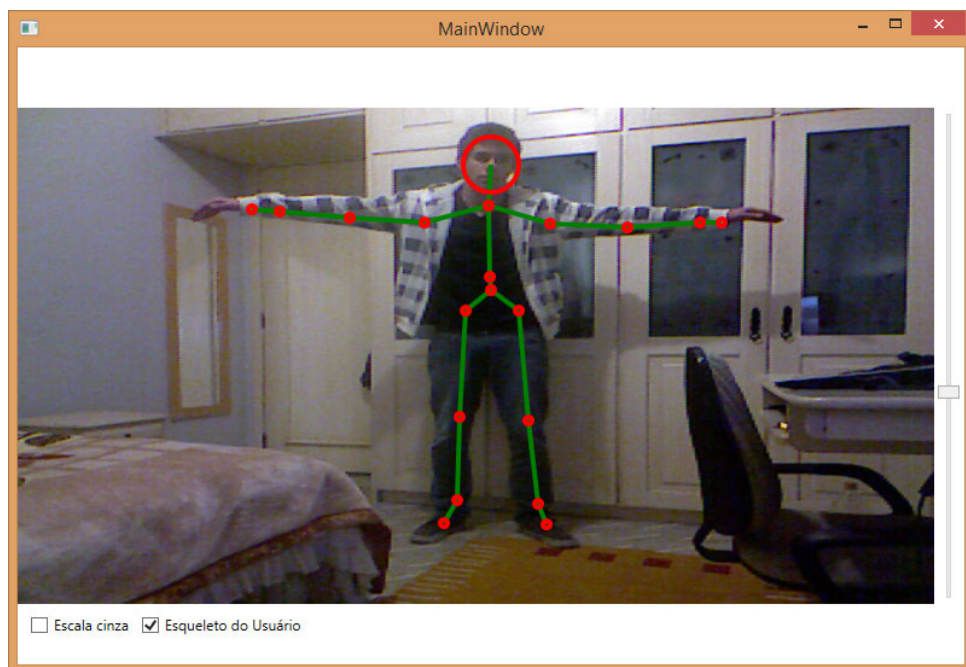
Figura 7.1: Identificando Pose T

Com toda a estrutura para rastreamento de poses pronta fica muito mais fácil criar a identificação de uma nova pose. Implementaremos uma nova pose para que possamos alterar a opção para exibir a imagem em escala cinza sem precisarmos tocar no computador. A pose em questão é comumente utilizada para pausar jogos no Xbox, mas nosso caso ela irá servir para alternar entre imagem comum e escala cinza, chamaremos esta pose de "pause", o usuário deve ficar na posição ilustrada na figura 7.2.
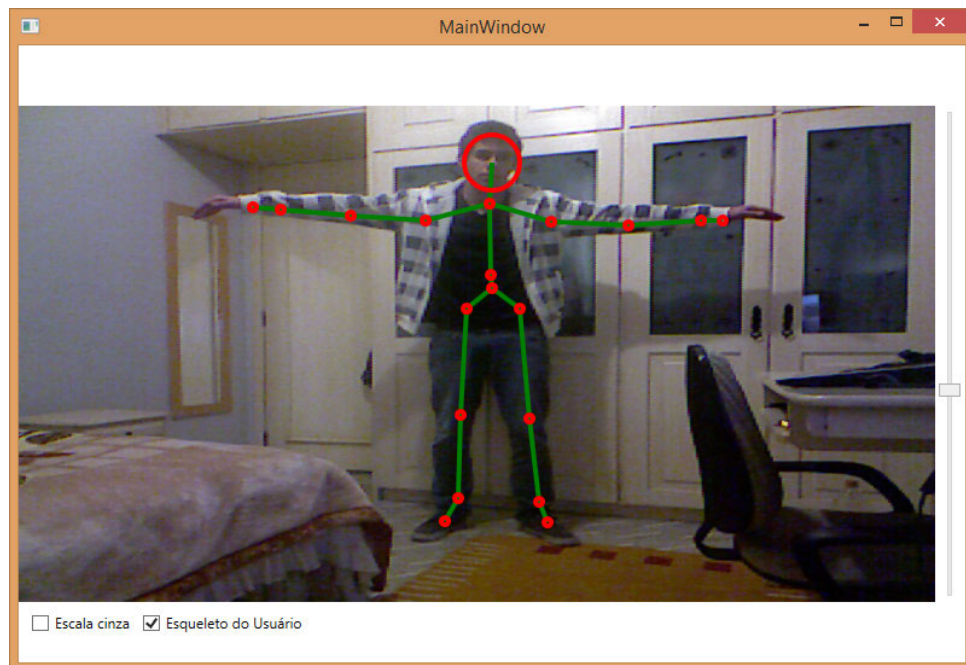
Figura 7.2: Pose de Pause

Na pose que implementamos anteriormente utilizamos apenas comparações entre as posições das articulações em seus eixos, mas há um outro tipo de abordagem para comparação de poses mais complexas, como por exemplo, a pose de pause que criaremos. Esta abordagem compara além da relação entre duas articulações, o ângulo entre três articulações, que pode definir, por exemplo, o quão aberto deve estar o braço do usuário, que é o que faremos nesta pose.

Para calcular o ângulo entre três articulações utilizaremos um método conhecido como **produto escalar**. Este método é utilizado para calcular ângulos entre dois vetores (V e W) em um espaço de três dimensões. Para fazer o cálculo é necessário aplicar a fórmula do produto escalar, ilustrada pela figura 7.3.

Figura 7.3: Fórmula do Produto Escalar

Apesar de parecer um cálculo pequeno é uma fórmula bastante complexa, iremos desenvolvê-la passo a passo para facilitar a compreensão.

Para gerarmos os vetores necessários para o cálculo é necessário utilizar as posições X, Y e Z das três articulações envolvidas, sendo que o vetor V deve ser gerado pelas articulações 1 e 2 e o vetor W deve ser gerado pelas articulações 2 e 3. Cada eixo dos vetores é calculado pela diferença entre os eixos das articulações envolvidas, mas é necessário elevar o resultado ao quadrado e calcular sua raiz quadrada, isso fará com que todo resultado se torne positivo, já que a distância entre dois pontos não deve ser negativa.

Implementaremos o produto escalar na classe `Util` de nosso projeto `AuxiliarKinect`, então criaremos um método privado para obter os vetores, conforme o código a seguir.

```
private static Vector4 CriarVetorEntreDoisPontos
 (Joint articulacao1, Joint articulacao2)
{
    Vector4 vetorResultante = new Vector4();

    vetorResultante.X = Convert.ToSingle(
     Math.Sqrt(
     Math.Pow( articulacao1.Position.X -
               articulacao2.Position.X, 2
             )));

    vetorResultante.Y = Convert.ToSingle(
     Math.Sqrt(
     Math.Pow( articulacao1.Position.Y -
               articulacao2.Position.Y, 2
             )));

    vetorResultante.Z = Convert.ToSingle(
     Math.Sqrt(
```

```
    Math.Pow( articulacao1.Position.Z -
              articulacao2.Position.Z, 2
            )));

    return vetorResultante;
}
```

Agora já conseguimos obter o valor dos vetores V e W descritos na fórmula. Precisamos calcular o produto dos vetores e o produto do módulo dos vetores. Para calcularmos o produto dos vetores soma-se o resultado do produto de cada coordenada dos vetores envolvidos, conforme a figura 7.4.

$$(Vx \, . \, Wx + Vy \, . \, Wy + Vz \, . \, Wz)$$

Figura 7.4: Produto dos Vetores

Vamos criar um método para calcular o resultado do produto de dois vetores, que também pode ser privado, pois utilizaremos apenas para o produto escalar que também será criado nesta classe. Seu método deve ficar similar ao código a seguir.

```
private static double ProdutoVetores
 (Vector4 vetorV, Vector4 vetorW)
{
    return vetorV.X * vetorW.X +
           vetorV.Y * vetorW.Y +
           vetorV.Z * vetorW.Z;
}
```

Para calcular o produto do módulo dos vetores é um pouco mais complicado. Para fazer isso é necessário multiplicar a raiz quadrada do resultado da soma do quadrado de cada eixo do vetor, conforme a fórmula ilustrada pela figura 7.5

$$\left(\sqrt{Vx^2 + Vy^2 + Vz^2}\right) . \left(\sqrt{Wx^2 + Wy^2 + Wz^2}\right)$$

Figura 7.5: Produto do Módulo dos Vetores

Também criaremos um método para calcular este resultado, conforme o código.

```
private static double
 ProdutoModuloVetores(Vector4 vetorV, Vector4 vetorW)
{
    return Math.Sqrt(
            Math.Pow( vetorV.X, 2) +
            Math.Pow( vetorV.Y, 2) +
            Math.Pow( vetorV.Z, 2))
            *
           Math.Sqrt(
            Math.Pow( vetorW.X, 2) +
            Math.Pow( vetorW.Y, 2) +
            Math.Pow(vetorW.Z, 2));
  }
```

Olhando na fórmula principal do cálculo você deve ter notado que há um cosseno elevado a menos um, esta operação é chamada de arco cosseno e está disponível na classe `Math` através do método `Acos`. Agora implementaremos o método principal para calcular o produto escalar, lembrando que o resultado do produto escalar será em radianos e o converteremos para graus, para fazer isso é necessário efetuar a multiplicação por 180 e dividir pelo número PI. Seu método deve ficar semelhante ao método a seguir.

```
public static double
 CalcularProdutoEscalar(Joint articulacao1,
                        Joint articulacao2,
                        Joint articulacao3)
    {
        Vector4 vetorV =
         CriarVetorEntreDoisPontos(articulacao1, articulacao2);

        Vector4 vetorW =
         CriarVetorEntreDoisPontos(articulacao2, articulacao3);

        double resultadoRadianos =
         Math.Acos(
         ProdutoVetores(vetorV, vetorW)
         /
         ProdutoModuloVetores(vetorV, vetorW));
```

```
        double resultadoGraus = resultadoRadianos * 180 / Math.PI;
        return resultadoGraus;
    }
```

Neste ponto já conseguimos calcular o produto escalar entre três articulações, então precisamos criar a classe que representa nossa pose para ativar a escala cinza. O método de validação desta pose deve utilizar o produto escalar para validar o ângulo entre o quadril esquerdo, ombro esquerdo e mão esquerda. Além disso, é necessário verificar se a mão está na mesma distância que o quadril e se ela está depois do cotovelo. Nossa classe ficará igual ao código a seguir.

```
public class PosePause : Pose
{
    public PosePause()
    {
        this.Nome = "PosePause";
        this.QuadroIdentificacao = 30;
    }

    protected override bool
     PosicaoValida(Microsoft.Kinect.Skeleton esqueletoUsuario)
    {
        const double ANGULO_ESPERADO = 25;
        double margemErroPosicao = 0.30;
        double margemErroAngulo = 10;


        Joint quadrilEsquerdo =
         esqueletoUsuario.Joints[JointType.HipLeft];

        Joint ombroEsquerdo =
         esqueletoUsuario.Joints[JointType.ShoulderLeft];

        Joint maoEsquerda =
         esqueletoUsuario.Joints[JointType.HandLeft];

        Joint cotoveloEsquerdo =
         esqueletoUsuario.Joints[JointType.ElbowLeft];

        double resultadoAngulo =
         Util.CalcularProdutoEscalar(quadrilEsquerdo,
```

```
            ombroEsquerdo, maoEsquerda);

        bool anguloCorreto =
            Util.CompararComMargemErro(margemErroAngulo,
            resultadoAngulo, ANGULO_ESPERADO);

        bool maoEsquerdaDistanciaCorreta =
            Util.CompararComMargemErro(margemErroPosicao,
            maoEsquerda.Position.Z, quadrilEsquerdo.Position.Z);

        bool maoEsquerdaAposCotovelo =
            maoEsquerda.Position.X < cotoveloEsquerdo.Position.X;

        bool maoEsquerdaAbaixoCotovelo =
            maoEsquerda.Position.Y < cotoveloEsquerdo.Position.Y;

        return anguloCorreto &&
               maoEsquerdaDistanciaCorreta &&
               maoEsquerdaAposCotovelo &&
               maoEsquerdaAbaixoCotovelo;
    }
}
```

Apenas com esta classe já é possível rastreá-la já que toda a implementação da estrutura de rastreamento já foi feita. Agora vamos voltar à nossa janela e alterar o método `InicilizarRastreadores` para que crie um rastreador também para a pose de pause, conforme o código.

```
private void InicializarRastreadores()
{
    rastreadores = new List<IRastreador>();

    Rastreador<PoseT> rastreadorPoseT = new Rastreador<PoseT>();
    rastreadorPoseT.MovimentoIdentificado += PoseTIdentificada;

    Rastreador<PosePause> rastreadorPosePause =
        new Rastreador<PosePause>();
    rastreadorPosePause.MovimentoIdentificado +=
        PosePauseIdentificada;
    rastreadores.Add(rastreadorPoseT);
    rastreadores.Add(rastreadorPosePause);
```

```
}

private void PosePauseIdentificada(object sender, EventArgs e)
{
    chkEscalaCinza.IsChecked = !chkEscalaCinza.IsChecked;
}
```

Com isso já podemos executar nossa aplicação e testar a pose em questão, conforme ilustra a figura 7.6
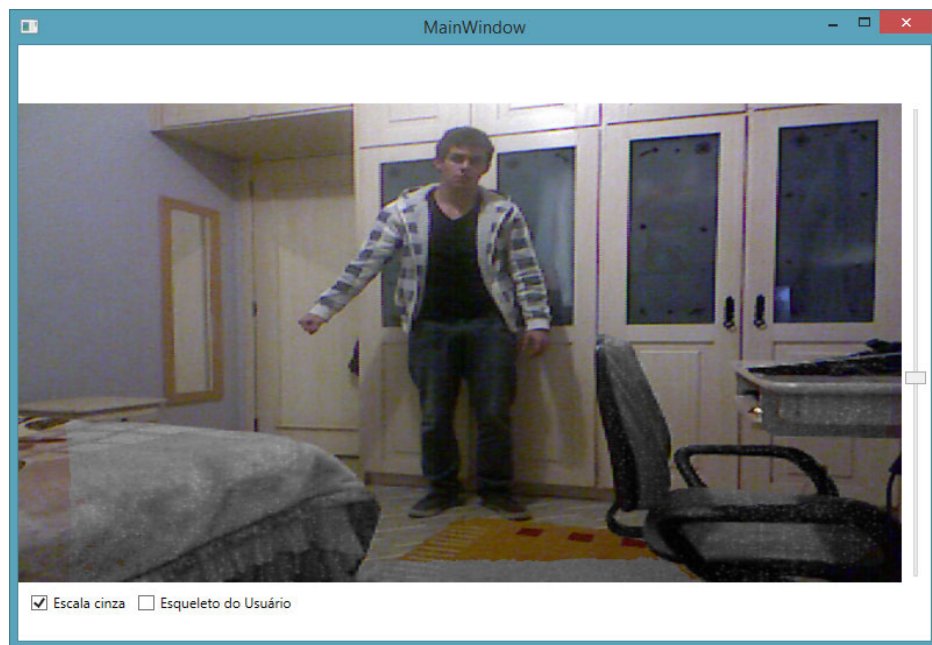
Figura 7.6: Identificando Pose T

O tempo até que esta pose seja detectada é de aproximadamente um segundo e, enquanto o usuário está parado nela, ele não recebe nenhum feedback da aplicação. Isso pode tornar a experiência prejudicada, pois o usuário não sabe está executando a pose de forma correta, então iremos fornecer uma forma de feedback ao usuário através do evento `MovimentoEmProgresso`.

Todos os objetos que herdam do tipo `Pose` já possuem a propriedade que indica o percentual de progresso, então basta criarmos uma forma de notificar essa informação ao usuário. Faremos isso através de uma barra de percentual, criaremos

esta barra somente quando o usuário estiver com a pose em progresso. Basicamente iremos obter a pose que é enviada por parâmetro no evento e criaremos dois retângulos, um para fazer o fundo da barra e outro para mostrar o progresso. Seu código deve ficar similar ao código a seguir.

```
private void
 PosePauseEmProgresso(object sender, EventArgs e)
{
    PosePause pose = sender as PosePause;

    Rectangle retangulo = new Rectangle();
    retangulo.Width = canvasKinect.ActualWidth;
    retangulo.Height = 20;
    retangulo.Fill = Brushes.Black;

    Rectangle poseRetangulo = new Rectangle();
    poseRetangulo.Width =
     canvasKinect.ActualWidth * pose.PercentualProgresso / 100;

    poseRetangulo.Height = 20;
    poseRetangulo.Fill = Brushes.BlueViolet;

    canvasKinect.Children.Add(retangulo);
    canvasKinect.Children.Add(poseRetangulo);
}
```

Apesar de não estar descrito no código anterior lembre-se de alterar o método `InicializarRastreadores` para interpretar o evento `MovimentoEmProgresso`, com isso você já pode testar e perceberá que quando você está na posição correta a barra de progresso é exibida no topo de nosso canvas, conforme ilustra a figura 7.7.
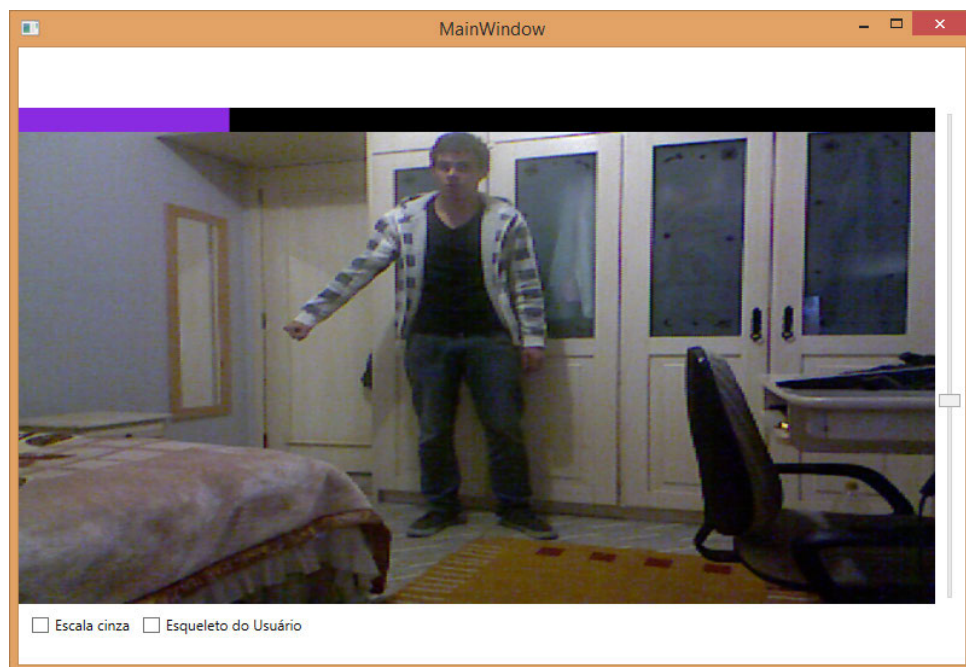
Figura 7.7: Feedback da Pose de Pause

Agora a usabilidade de nosso aplicativo já melhorou bastante, pois o usuário percebe quando a pose está sendo executada de forma correta e quanto tempo é necessário até que ela se torne ativa.

Criaremos o reconhecimento de um gesto para que isso ative uma função na aplicação. Para fins didáticos iremos implementar o gesto de aceno, comumente utilizado para despedir-se de pessoas. Mas antes de implementarmos é necessário finalizarmos a nossa estrutura base para detecção de movimentos.

7.3 Finalizando a Estrutura Base para Detectar Movimentos

Nesta etapa iremos finalizar nossa estrutura para detecção de movimentos implementando a parte para detecção de gestos. Existem diversas formas para a detecção de gestos, as mais conhecidas são: redes neurais artificiais, comparação a um gesto previamente armazenado e detecção por algoritmo especializado.

O processo de detecção utilizando a técnica de inteligência artificial conhecida

por redes neurais artificiais é utilizado em diferentes tipos de sensores, incluindo o Kinect. Esta é uma forma de detecção não trivial e complexa, não indicada para aplicações simples. Com uma rede bem treinada, o rastreamento e detecção de um gesto é classificado e feito com boa precisão. Porém o processo de criação e treinamento de uma rede neural pode ser bastante custoso e poderá ser utilizado em somente um gesto específico.

O processo de detecção utilizando comparação a um exemplo prévio consiste em registrar os movimentos de um usuário através de um software especializado em captura. Durante a execução de sua aplicação, será necessário utilizar de APIs para interagir com o arquivo previamente gravado e comparar em tempo real.

Os dois exemplos acima fogem do escopo deste livro, então utilizaremos a abordagem utilizando algoritmo especializado. Esta é a forma menos complexa e que possui um bom resultado se o algoritmo for implementado da forma correta. Esta abordagem é utilizada geralmente em gestos mais simples.

Iremos utilizar algoritmos especializados, mas além disso, é necessário escolher qual algoritmo iremos criar. Neste caso, para reaproveitarmos o código, iremos utilizar um algoritmo conhecido como **keyframes**. Neste algoritmo podemos interpretar um gesto como uma série de poses a serem reconhecidas em seguida em determinado espaço de tempo, ou seja, um gesto é uma coleção de poses que são executadas uma após a outra e todas elas possuem uma margem de tempo para execução. Caso essas validações sejam cumpridas o gesto é aceito.

Para cada quadro chave do gesto é necessário termos uma pose, que é a pose que o usuário deve estar e dois valores para controlar a quantidade mínima e máxima de quadros que devem ter se passado desde que o quadro chave anterior foi aceito. Estas propriedades são para identificar a velocidade com que o usuário deve fazer as poses contidas no gesto.

Vamos abrir novamente o projeto `AuxiliarKinect` e criaremos no namespace `Movimentos` a classe `GestoQuadroChave`, essa classe irá identificar cada quadro chave de um gesto. Conforme descrevemos no parágrafo anterior, esta classe deve conter a pose que o usuário deverá estar nesse quadro chave e os tempos limites inferior e superior para que este quadro chave seja reconhecido após o quadro chave anterior. Sua classe deve ficar semelhante ao código a seguir.

```
public class GestoQuadroChave
{
    public int QuadroLimiteInferior { get; private set; }
    public int QuadroLimiteSuperior { get; private set; }
```

```
    public Pose PoseChave { get; private set; }

    public GestoQuadroChave
     (Pose poseChave, int limiteInferior, int limiteSuperior)
    {
        this.PoseChave = poseChave;
        this.QuadroLimiteInferior = limiteInferior;
        this.QuadroLimiteSuperior = limiteSuperior;
    }
}
```

Agora vamos criar a classe abstrata que será base para todos os gestos de nossa aplicação, ela se chamará `Gesto` e também deverá ser criada no namespace `Movimentos`. A classe `Gesto` assim como a classe `Pose` deve herdar de `Movimento` e sobrescrever o método `Rastrear`, que por enquanto ficará em branco, como fizemos anteriormente.

No caso de gestos é necessário possuir uma coleção de quadros chave, pois este será o princípio para a detecção de qualquer gesto implementado. Além disso, teremos que criar uma propriedade para armazenar em tempo de execução qual o quadro chave atual do gesto que o usuário se encontra. Por fim, para podermos passar algum *feedback* ao usuário, devemos ter um contador de quadros chaves e duas propriedades para que seja possível obter a quantidade de etapas total do gesto e o percentual de etapas que já foram concluídas, conforme o código abaixo.

```
public abstract class Gesto : Movimento
{
    protected LinkedList<GestoQuadroChave> QuadrosChave { get; set; }
    protected LinkedListNode<GestoQuadroChave> QuadroChaveAtual
                                                {get;set;}
    private int contadorEtapas;

    public int QuantidadeEtapas
    {
        get
        {
            return QuadrosChave.Count;
        }
    }

    public int PercentualEtapasConcluidas
```

```
    {
        get
        {
            return contadorEtapas * 100 / QuantidadeEtapas;
        }
    }

    public override EstadoRastreamento Rastrear
     (Skeleton esqueletoUsuario)
    {

    }
}
```

Observe que armazenamos os quadros chave em uma coleção do tipo `LinkedList`. Faremos isso para facilitar a tarefa de encontrar a próxima etapa da lista, já que nesta coleção cada elemento conhece seu sucessor e antecessor.

Temos nossa base para criação de gestos, que utilizará fortemente nossa estrutura já pronta, tanto para detectar as poses de seus quadros chave, quanto para utilizar os rastreadores para executarem sua detecção. Criaremos dentro da pasta `Movimentos`, a pasta `Gestos` assim como fizemos com as poses.

A pasta `Gestos` será organizada de forma diferente da pasta `Poses`. No caso de uma pose, apenas uma classe era utilizada para a criação de uma pose para a aplicação, já no caso dos gestos é necessária a criação de uma classe que herde da classe `Gesto` e de todas as poses que são utilizadas em seus quadros chave. Para que a estrutura do projeto não fique confusa iremos criar a pasta `Aceno` dentro da pasta `Gestos` e nela devem estar todas as classes utilizadas para o gesto de aceno que iremos implementar.

Nosso gesto de aceno será composto por sete quadro chaves, porém como ilustra a figura 7.8 teremos que criar apenas três poses, pois elas são reaproveitadas ao decorrer dos gestos.
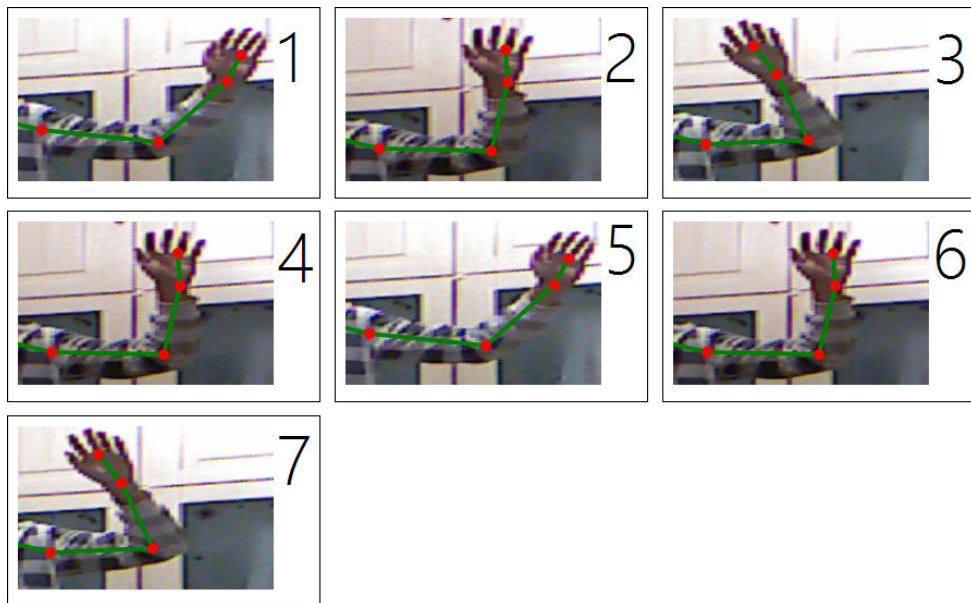
Figura 7.8: Quadros Chave do Gesto Aceno

Diferente das poses que criamos anteriormente, as poses que compõem o gesto de aceno são bem mais simples. Começaremos criando a pose para identificar os quadros chaves um e cinco da figura ilustrada anteriormente. Nesses quadros é necessário verificar se a mão do usuário está mais longe do corpo e mais alta que seu cotovelo. Esta classe será implementada como qualquer outra pose, mas iremos incluí-la no namespace `Gestos.Aceno`, já que ela será utilizada exclusivamente para o gesto. Seu código deve ficar similar ao código a seguir.

```
public class AcenoMaoAposCotovelo : Pose
{
    protected override bool PosicaoValida
     (Skeleton esqueletoUsuario)
    {
        Joint maoDireita =
         esqueletoUsuario.Joints[JointType.HandRight];

        Joint cotoveloDireito =
         esqueletoUsuario.Joints[JointType.ElbowRight];
```

```
        bool maoDireitaAposCotovelo =
         maoDireita.Position.X > cotoveloDireito.Position.X;

        bool maoDireitaAcimaCotovelo =
         maoDireita.Position.Y > cotoveloDireito.Position.Y;

        return maoDireitaAcimaCotovelo && maoDireitaAposCotovelo;
    }
}
```

Note que não há diferença alguma da criação de uma pose que será utilizada como quadro chave e de uma pose independente. Agora faremos uma classe bastante similar a ela, mas que irá validar se a mão do usuário está mais próxima do corpo do usuário ao invés de mais longe, este caso é utilizado nos quadros chave três e sete do aceno.

```
public class AcenoMaoAntesCotovelo : Pose
{
    protected override bool PosicaoValida
     (Skeleton esqueletoUsuario)
    {
        Joint maoDireita =
         esqueletoUsuario.Joints[JointType.HandRight];

        Joint cotoveloDireito =
         esqueletoUsuario.Joints[JointType.ElbowRight];

        bool maoDireitaAntesCotovelo =
         maoDireita.Position.X < cotoveloDireito.Position.X;

        bool maoDireitaSobreCotovelo =
         maoDireita.Position.Y > cotoveloDireito.Position.Y;

        return maoDireitaSobreCotovelo && maoDireitaAntesCotovelo;
    }
}
```

Por fim, iremos implementar a pose utilizada nos quadros chave dois, quatro e seis. Nesta pose, é necessário validar se a mão do usuário está mais alta e alinhada verticalmente com seu cotovelo, lembrando que em comparações de igualdade é necessário utilizar margem de erro na comparação, conforme o código a seguir.

```
public class AcenoMaoSobreCotovelo : Pose
{
    protected override bool PosicaoValida
     (Skeleton esqueletoUsuario)
    {
        double margemErro = 0.30;
        Joint maoDireita =
         esqueletoUsuario.Joints[JointType.HandRight];

        Joint cotoveloDireito =
         esqueletoUsuario.Joints[JointType.ElbowRight];

        bool maoDireitaAntesCotovelo =
         maoDireita.Position.X > cotoveloDireito.Position.X;

        bool maoDireitaSobreCotovelo =
         Util.CompararComMargemErro(margemErro,
         maoDireita.Position.Y, cotoveloDireito.Position.Y);

        return maoDireitaSobreCotovelo && maoDireitaAntesCotovelo;
    }
}
```

Com isso já temos todas as poses que serão utilizadas para rastrear e identificar um aceno do usuário! Agora criaremos a classe `Aceno` que deve herdar da classe `Gesto`. Iremos criar um método chamado `InicializaQuadrosChave`, que será utilizado para instanciar a nossa lista encadeada que armazena os quadros chave e preenchê-la com os quadros chave que compõe nosso gesto. Além disso, iremos inicializar as propriedades `Nome`, `ContadorQuadros` e `QuadroChaveAtual`.

Ao herdar a classe `Gesto`, torna-se necessária a implementação do método `PosicaoValida`, que deve recuperar o retorno do método `Rastrear` da pose que está em seu quadro atual e retornar se a pose que está no quadro atual foi reconhecida, conforme o código abaixo.

```
public class Aceno : Gesto
{
    public Aceno()
    {
        InicializaQuadrosChave();
```

```
        Nome = "Aceno";
        ContadorQuadros = 0;
        QuadroChaveAtual = QuadrosChave.First;
    }

    private void InicializaQuadrosChave()
    {
        QuadrosChave = new LinkedList<GestoQuadroChave>();
        QuadrosChave.AddFirst(
         new GestoQuadroChave(new AcenoMaoAposCotovelo(), 0, 0)
         );

        QuadrosChave.AddLast(
         new GestoQuadroChave(new AcenoMaoSobreCotovelo(), 1, 25)
         );

        QuadrosChave.AddLast(
         new GestoQuadroChave(new AcenoMaoAntesCotovelo(), 1, 25)
         );

        QuadrosChave.AddLast(
         new GestoQuadroChave(new AcenoMaoSobreCotovelo(), 1, 25)
         );

        QuadrosChave.AddLast(
         new GestoQuadroChave(new AcenoMaoAposCotovelo(), 1, 25)
         );

        QuadrosChave.AddLast(
         new GestoQuadroChave(new AcenoMaoSobreCotovelo(), 1, 25)
         );

        QuadrosChave.AddLast(
         new GestoQuadroChave(new AcenoMaoAntesCotovelo(), 1, 25)
         );
    }

    protected override bool PosicaoValida(Skeleton esqueletoUsuario)
    {
        EstadoRastreamento estado =
         QuadroChaveAtual.Value.PoseChave.Rastrear(esqueletoUsuario);
```

```
        return estado == EstadoRastreamento.Identificado;
    }
}
```

Note que inserimos como tempo limite inferior o valor de um quadro. Com o gesto implementado desta forma não haverá limites para a velocidade máxima do usuário executar o gesto. Caso deseja controlar melhor a velocidade de execução do gesto, controle essas propriedades a seu gosto.

Agora para finalizar a estrutura base de gestos precisamos retornar à classe `Gesto` e implementar o método `Rastrear`. Diferente da pose, a implementação para rastreio de um gesto é bastante complexa e irá exigir um pouco mais de atenção nessa parte. Para facilitar a compreensão foi feito um fluxograma das regras ilustrado pela figura 7.9.
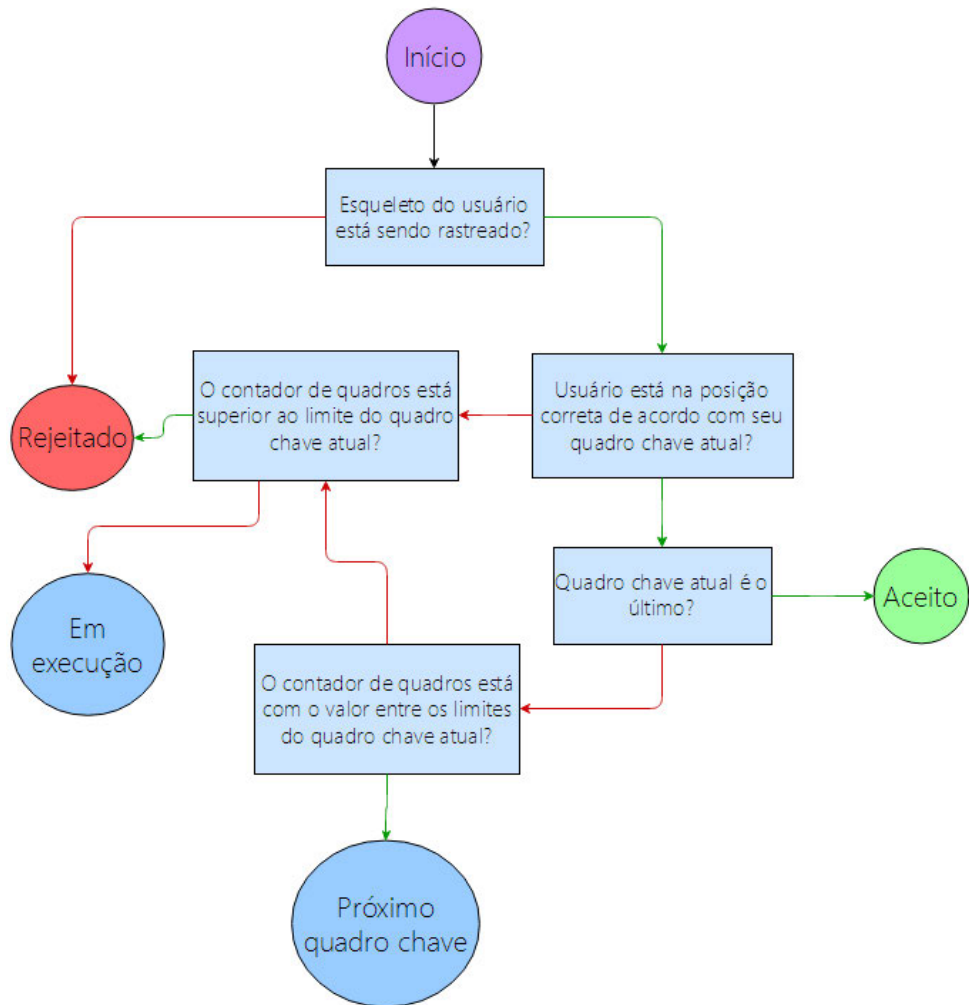
Figura 7.9: Fluxograma de Regras para Rastrear um Gesto

Antes de começarmos a definir as regras para o método `Rastrear` iremos criar os métodos que alteram o estado do rastreamento e um atributo interno para controlar o novo estado que o gesto irá adquirir após a chamada para o método rastrear. O primeiro método a ser criado será chamado de `ReiniciarRastreamento`. Ele será invocado sempre que for necessário reiniciar o rastreamento do gesto, e deverá reiniciar o quadro chave atual e a contagem de quadros e etapas. Além disso, o novo estado adquirido deve ser `NaoIdentificado`, conforme o código.

```
private void ReiniciarRastreamento()
{
    ContadorQuadros = 0;
    QuadroChaveAtual = QuadrosChave.First;
    contadorEtapas = 0;
    novoEstado = EstadoRastreamento.NaoIdentificado;
}
```

Note que, apesar de não ter sido ilustrado em nenhum código, o campo `novoEstado` deve ser declarado no escopo da classe. Após isso, criaremos o método para avançar para o próximo quadro chave, conforme o código a seguir.

```
private void ProximoQuadroChave()
{
    novoEstado = EstadoRastreamento.EmExecucao;
    ContadorQuadros = 0;
    QuadroChaveAtual = QuadroChaveAtual.Next;
    contadorEtapas++;
}
```

Por fim, criaremos um método para permanecer em fase de rastreamento sem alterar o quadro chave, segue o código.

```
private void PermanecerRastreando()
{
    ContadorQuadros++;
    novoEstado = EstadoRastreamento.EmExecucao;
}
```

Agora que já temos os métodos para alterarmos o estado de rastreamento do gesto vamos implementar o método principal, seguindo as regras ilustradas pelo fluxograma, conforme o código.

```
public override EstadoRastreamento Rastrear
 (Skeleton esqueletoUsuario)
{
    if (esqueletoUsuario != null)
    {
        if (PosicaoValida(esqueletoUsuario))
        {
            novoEstado = EstadoRastreamento.EmExecucao;
```

```
            if (QuadroChaveAtual.Value == QuadrosChave.Last.Value)
                novoEstado = EstadoRastreamento.Identificado;
            else
            {
                if (ContadorQuadros >=
                    QuadroChaveAtual.Value.QuadroLimiteInferior &&
                    ContadorQuadros <=
                    QuadroChaveAtual.Value.QuadroLimiteSuperior)

                    ProximoQuadroChave();
                else
                    if ( ContadorQuadros <
                         QuadroChaveAtual.Value.QuadroLimiteInferior)

                        PermanecerRastreando();

                    else if (ContadorQuadros >
                             QuadroChaveAtual.Value.QuadroLimiteSuperior)

                        ReiniciarRastreamento();
            }
        }
        else
            if (ContadorQuadros <
                QuadroChaveAtual.Value.QuadroLimiteSuperior)

                ReiniciarRastreamento();
            else
                PermanecerRastreando();
    }
    else
        ReiniciarRastreamento();

    return novoEstado;
}
```

Agora já podemos voltar para nossa aplicação, criar o rastreador para nosso gesto de aceno e tomar uma ação quando o gesto for reconhecido!

7.4 Utilizando Gestos em Nossa Aplicação

Com toda a estrutura já preparada, a alteração em nossa aplicação será bastante pequena — basta alterarmos o método `InicializarRastreadores` para que ele passe a criar também um rastreador para o gesto de aceno e implementarmos um método para executar uma ação quando o evento `MovimentoIdentificado` for disparado.

Vamos começar refatorando o método que inicializa os rastreadores de nossa aplicação para adicionarmos mais um rastreador em nossa lista `rastreadores`. Basta seguir o mesmo padrão dos já implementados rastreadores, como no código a seguir.

```
private void InicializarRastreadores()
{
    rastreadores = new List<IRastreador>();

    Rastreador<PoseT> rastreadorPoseT =
     new Rastreador<PoseT>();
    rastreadorPoseT.MovimentoIdentificado += PoseTIdentificada;

    Rastreador<PosePause> rastreadorPosePause =
     new Rastreador<PosePause>();
    rastreadorPosePause.MovimentoIdentificado += PosePauseIdentificada;
    rastreadorPosePause.MovimentoEmProgresso += PosePauseEmProgresso;

    Rastreador<Aceno> rastreadorAceno =
     new Rastreador<Aceno>();
    rastreadorAceno.MovimentoIdentificado += AcenoIndentificado;

    rastreadores.Add(rastreadorPoseT);
    rastreadores.Add(rastreadorPosePause);
    rastreadores.Add(rastreadorAceno);
}

private void AcenoIndentificado(object sender, EventArgs e)
{

}
```

No código ilustrado anteriormente, já foi criado o método que é disparado no executar do evento de identificação do gesto de aceno, mas ainda não há nenhuma

ação de fato. Iremos fazer com que a aplicação seja fechada, sempre que o gesto de aceno de despedida for reconhecido. Para que a aplicação seja fechada, utilize a linha de código `Application.Current.Shutdown();`.

Abra sua aplicação e teste! Perceba que nossa aplicação já possui diferentes formas de interações naturais para ativar algumas funções, utilizando poses e gestos.

Capítulo 8

Interagindo com a aplicação através do KinectInteractions

Há um pacote de funcionalidades chamado `KinectInteractions`, que foi lançado na atualização 1.7 do SDK, porém ele não é disponibilizado junto com o SDK padrão — para acessá-lo é necessário utilizar o Kinect Toolkit. Através dele, é possível detectar uma série de comportamentos relacionados às mãos do usuário e também são disponibilizados diversos componentes de tela que devem ser utilizados para criar uma interface adequada ao Kinect. Todas estas funcionalidades e componentes foram criados para facilitar e melhorar a navegabilidade entre as aplicações que utilizam o sensor como forma de interação.

Parte dessas funcionalidades é disponibilizada através de um fluxo de interação, que será apresentado neste capítulo e iremos notar que ele é um pouco diferente dos fluxos mostrados anteriormente. Este tipo de fluxo não é uma propriedade da classe `KinectSensor` e sim um objeto que deve ser criado independente, devido ao fato de que o `KinectInteractions` não faz parte do SDK padrão.

8.1 Principais Conceitos

Dentro do `KinectInteractions` há uma série de conceitos que precisam ser conhecidos para podermos avançar com nossa aplicação. É importante ter boa familiaridade com eles pra podermos aproveitar ao máximo todo seu recurso.

Rastreamento Avançado da Mão

Como já vimos anteriormente, através do fluxo de esqueleto podemos rastrear diversas articulações do usuário, entre elas suas mãos. O `KinectInteractions` fornece uma forma mais avançada para rastrearmos a mão do usuário, para isso é necessário que ambos os fluxos esqueleto e profundidade já estejam habilitados. Com este tipo de rastreamento é possível identificar mais informações além do posicionamento. Estas informações são: identificar se o usuário está com a mão fechada, identificar se o usuário estava com a mão fechada e agora ele a abriu e identificar se o usuário está fazendo o gesto de pressionar, que é utilizado geralmente para ativar a função de botões ou tiles.

Fluxo de Interação

O fluxo de interação provê um fluxo contínuo de quadros de interação, semelhante ao modelo já visto nos outros fluxos, porém este necessita de informações do fluxo de esqueleto e do fluxo de profundidade. Cada quadro de interação provê informações relacionadas ao rastreamento avançado da mão e também é possível obter qual o componente de tela que a mão do usuário está alvejando, tornando a interação com a tela algo mais simples.

Diferente dos outros fluxos, não há apenas uma classe chamada `InteractionStream`. Além dela também existe uma interface chamada `IInteractionClient`, que deve ser implementada por componentes que podem ser interativos através do Kinect. Veremos como implementá-la ao longo deste capítulo.

Controles de Tela

Há uma série de controles de tela que já implementam a interface `IInteractionClient`, eles podem ser utilizados para facilitar as tarefas mais comuns, como por exemplo, um painel que exibe o ponteiro na posição da mão do usuário ou um botão para ser pressionado. Todos estes clientes de interação devem estar dentro de um componente conhecido como `KinectRegion`, que

é um tipo de painel que permite criar um vínculo com o sensor Kinect e através dele é possível interagir com os controles de tela que implementam a interface `IInteractionClient`.

8.2 Melhorando Nossa Aplicação com os Controles do KinectInteractions

A primeira coisa a fazer será melhorar a usabilidade de nossa aplicação para podermos utilizá-la com o Kinect de forma mais natural. Nos guias para desenvolvimento de aplicações naturais fornecidos pela própria Microsoft sempre é aconselhado oferecer mais de uma forma para executar a mesma funcionalidade, então é isso que iremos fazer. Já conseguimos ativar as funcionalidades "Escala Cinza" e "Esqueleto do Usuário" através de poses, mas os controles de tela que ativam isso necessitam de mouse e/ou teclado. Agora iremos alterar estes componentes para podermos ativá-los com o Kinect.

A primeira coisa a fazer é ter certeza de que as DLLs *Microsoft.Kinect.Toolkit*, *Microsoft.Kinect.Toolkit.Controls* e *Microsoft.Kinect.Toolkit.Interactions* estejam sendo referenciadas no projeto, além disso, é necessário copiar para as pastas *~\bin\Debug* e *~\bin\Release* a DLL *KinectInteractions_170_64* ou a DLL *KinectInteractions_170_32* dependendo da arquitetura de seu computador, x64 ou x86 respectivamente.

Com todas as referências e DLLs copiadas podemos começar nossa implementação. Como já dito antes, o foco desta alteração é modificar os controles de tela para que eles possam ser utilizados de forma natural com o Kinect e não mais através de mouse e teclado. Infelizmente nos controles nativos não há um substituto para o `CheckBox`, então criaremos um tipo de botão que é comumente chamado de `ToggleButton`, este tipo de botão possui estado de pressionado e não pressionado, funcionando de forma similar a um `CheckBox`.

Para fazermos isso devemos criar uma nova classe em nosso namespace `Auxiliar`, ela se chamará `KinectToggleButton` e deve herdar de um componente já presente no *Toolkit*, chamado de `KinectTileButton`. Esta classe deve conter uma propriedade para identificar se o botão está ou não pressionado e deve implementar o evento `Click` de sua classe base para alterar o estado dessa propriedade. Para identificação visual, alteraremos também a propriedade Background deste componente, de acordo com seu estado.

```
public class KinectToggleButton : KinectTileButton
{
```

```
    public bool IsChecked { get; set; }

    public KinectToggleButton()
    {
        Click += AlterarEstado;
        this.Background = Brushes.RoyalBlue;
    }

    private void AlterarEstado
     (object sender, System.Windows.RoutedEventArgs e)
    {
        IsChecked = !IsChecked;
        if (IsChecked)
            this.Background = Brushes.DarkBlue;
        else
            this.Background = Brushes.RoyalBlue;
    }
}
```

É bastante simples implementar este componente, pois grande parte das funcionalidades já são herdadas da classe `KinectTileButton`. Agora podemos voltar para nossa janela principal para alterarmos o seu layout. A primeira coisa a fazer é garantir que os namespaces de nosso componente `KinectToggleButton` e dos componentes padrão do *Interactions* estejam sendo importados em nossa janela. Para fazer isso, devemos incluí-los na declaração da janela no arquivo XAML, conforme o trecho de código.

```
<Window x:Class="Interacao.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:k="http://schemas.microsoft.com/kinect/2013"
    xmlns:my="clr-namespace:Interacao.Auxiliar"
    Title="MainWindow" Height="800" Width="1000"
    WindowState="Maximized">
```

Os namespaces do *KinectInteractions* e de nosso componente recebem os apelidos de *k* e *my* respectivamente. Note que, além de incluir os namespaces, alteramos o estado da janela para que ela seja aberta no modo maximizado, isso porque o painel inferior que possui os controles terá de aumentar de tamanho, pois a precisão do cursor utilizando o Kinect é menor que a do mouse, então você deve alterar o valor da propriedade `Height` do terceiro `RowDefinitions` do Grid de 50 para 180.

Agora na janela temos que circundar o StackPanel que os componentes estavam sendo inseridos por um KinectRegion, além disso precisamos alterar o tipo dos componentes de CheckBox para my:KinectToggleButton, conforme o trecho de código.

```
<k:KinectRegion Name="kinectRegion" Grid.Row="2" Grid.ColumnSpan="2">
    <StackPanel  Orientation="Horizontal" >

        <my:KinectToggleButton x:Name="btnEscalaCinza"
         Content="Escala Cinza" Height="130" Foreground="White"/>

        <my:KinectToggleButton x:Name="btnEsqueletoUsuario"
         Content="Esqueleto do Usuário" Height="130"
         FontSize="20" Foreground="White"/>

        </StackPanel>
</k:KinectRegion>
```

Precisamos alterar agora nossa classe referente à janela, e alterar todas as antigas referências aos CheckBoxes para nossos KinectToggleButtons, após fazer isso você já pode executar a aplicação para testá-la. Seu resultado será semelhante ao ilustrado pela figura 8.1.
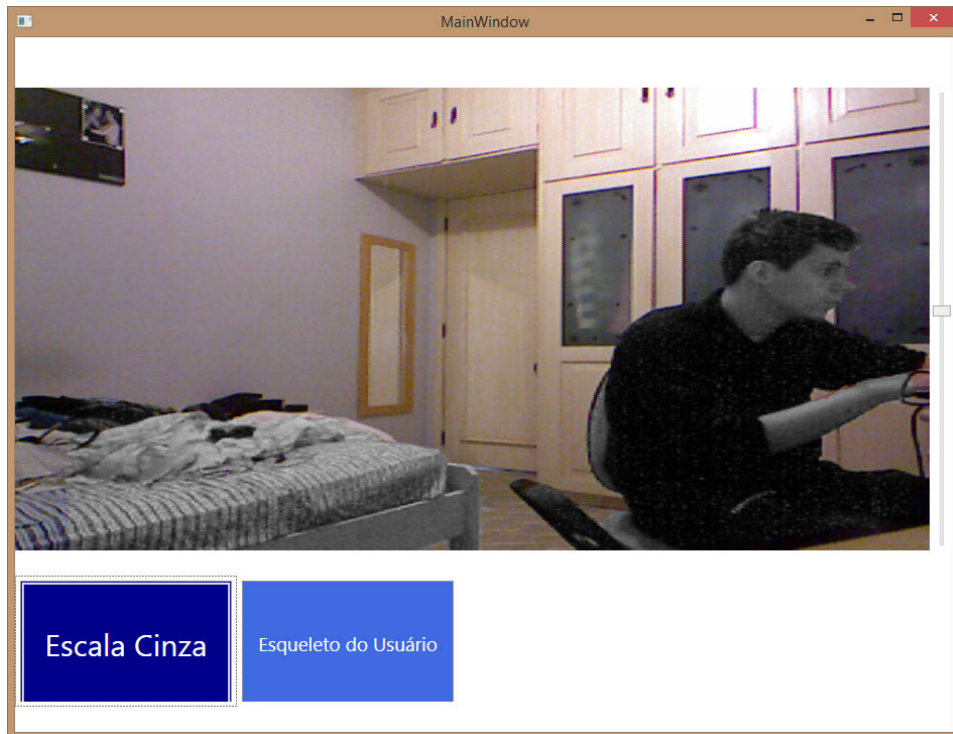
Figura 8.1: Kinect Toggle Buttons

Ao clicar com o mouse no botão, ele funcionará normalmente, mas ainda não há formas de interagir com eles através do Kinect, isso porque ainda não vinculamos o sensor Kinect ao nosso `KinectRegion`. No capítulo anterior, vinculamos o evento de identificação do gesto de aceno a um método que fecha a aplicação. Iremos alterar este comportamento, para que, quando o usuário execute o gesto de aceno, o `KinectRegion` receba o objeto do `KinectSensor`. Desta forma, antes de interagirmos com os botões é necessário executar o gesto de aceno. Para fazer isso vamos alterar o método o método `AcenoIndentificado` conforme código.

```
private void AcenoIndentificado(object sender, EventArgs e)
{
    if (kinectRegion.KinectSensor == null)
        kinectRegion.KinectSensor = kinect;
}
```

Ao fazer isso já conseguiremos interagir com os botões utilizando o Kinect,

mas ainda não temos a possibilidade de desativarmos a interação com os controles. Para podermos fazer isso, vamos adicionar ao StackPanel um controle do tipo KinectTileButton, que servirá para desligar o vínculo entre o sensor e o componente KinectRegion. Com essa alteração, o StackPanel dos componentes já existentes deve ficar semelhante ao a seguir.

```
<StackPanel  Orientation="Horizontal" >

    <my:KinectToggleButton x:Name="btnEscalaCinza"
     Content="Escala Cinza" Height="130" Foreground="White"/>

    <my:KinectToggleButton x:Name="btnEsqueletoUsuario"
     Content="Esqueleto do Usuário" Height="130"
     FontSize="20" Foreground="White"/>

    <k:KinectTileButton Content="Desligar" Height="130" Width="150"
     Foreground="White" Click="btnVoltarClick"/>

</StackPanel>
```

Perceba que declaramos o componente junto com os demais, e que já vinculamos um método para seu evento de Click. Este método está definido na classe que representa nossa janela para desvincular o sensor de nosso painel, conforme o código.

```
private void btnVoltarClick(object sender, RoutedEventArgs e)
{
    if (kinectRegion.KinectSensor != null)
        kinectRegion.KinectSensor = null;
}
```

Com isso já temos a possibilidade de interagir com a aplicação através do Kinect e os controles de tela disponibilizados no Interactions, mas ainda falta mantermos a funcionalidade para fecharmos a aplicação. Para isso, circundaremos novamente o StackPanel, mas desta vez com o componente DockPanel. Este painel permite utilizarmos um layout baseado em bordas e utilizando este recurso faremos com que o novo KinectTileButton, que será utilizado para fecharmos a aplicação, fique sempre preso à borda da direita, distanciando-se dos demais botões e evitando que ele seja pressionado pelo usuário por engano. O código abaixo ilustra como deve ficar a hierarquia de seu KinectRegion.

```
<k:KinectRegion Name="kinectRegion" Grid.Row="2" Grid.ColumnSpan="2">
    <DockPanel>

        <k:KinectTileButton Content="Fechar"
         Height="130" Width="150" Foreground="White"
         DockPanel.Dock="Right" Click="btnFecharClick"/>

        <StackPanel  Orientation="Horizontal" >

            <my:KinectToggleButton x:Name="btnEscalaCinza"
             Content="Escala Cinza" Height="130"
             Foreground="White"/>

            <my:KinectToggleButton x:Name="btnEsqueletoUsuario"
             Content="Esqueleto do Usuário" Height="130"
             FontSize="20" Foreground="White"/>

            <k:KinectTileButton Content="Desligar"
             Height="130" Width="150"
             Foreground="White" Click="btnVoltarClick"/>

        </StackPanel>
    </DockPanel>
</k:KinectRegion>
```

O `KinectTileButton` para fechar a aplicação também deve ter um método vinculado a seu evento de `Click` e seu método deve fazer a mesma coisa que fazíamos na detecção do gesto de aceno, conforme o código.

```
private void btnFecharClick(object sender, RoutedEventArgs e)
{
    Application.Current.Shutdown();
}
```

Com isso, você já pode testar novamente sua aplicação, o resultado deve ser semelhante ao ilustrado pela figura 8.2.
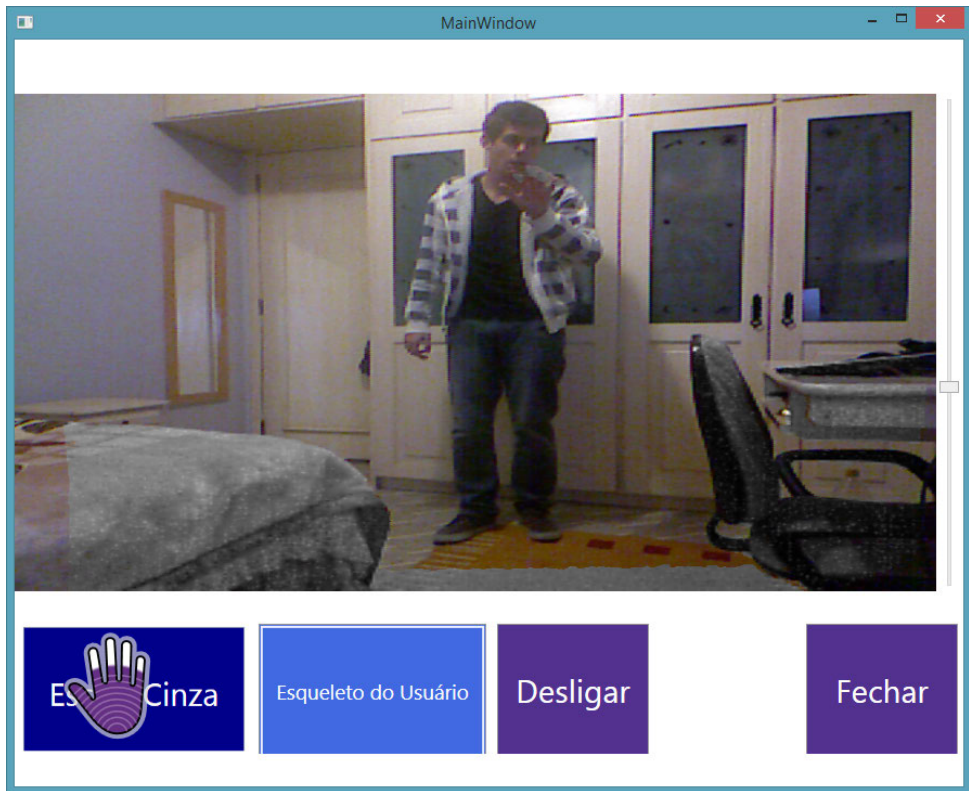
Figura 8.2: Aplicação utilizando o Interactions

Para ativar os botões, é necessário fazer o gesto de pressionar, para fazer isso basta esticar a mão como a figura anterior ilustra.

8.3 Utilizando o Fluxo de Interação

Já vimos como é possível utilizar os controles disponíveis para melhorar a usabilidade de nossa aplicação com o Kinect, agora veremos como utilizar o próprio fluxo de interação. Vale lembrar que, para que este fluxo seja processado, é necessário que os fluxos de profundidade e esqueleto estejam habilitados também.

Como proposta para uma nova funcionalidade iremos fazer com que seja possível para o usuário criar desenhos na imagem que está sendo exibida na aplicação, ou seja, além da imagem da câmera, será permitido ao usuário criar desenhos utilizando usas mãos como pincéis. Mas para que a mão se torne de fato um pincel o

usuário terá de fechá-la — para reconhecer o estado da mão do usuário (aberta ou fechada) iremos aproveitar o recurso disponibilizado pelo fluxo interação.

Você já deve ter percebido que para desenhar elementos utilizamos o componente `Canvas`, mas este componente não tem integração com o fluxo de interação, então criaremos uma nova classe que herde da classe do componente `Canvas` e implemente a interface `IInteractionClient`, criando assim um novo `Canvas` que terá todas as funcionalidades do original e mais a compatibilidade com o fluxo de interação.

Vamos criar no namespace `Auxiliar` a classe `CanvasInteracao`, seguindo o que foi dito no parágrafo anterior. Ao implementar a interface `IInteractionClient`, será criado o método `GetInteractionInfoAtLocation` em sua classe. Este método deve retornar um objeto do tipo `InteractionInfo`, que descreve os tipos de interação que podem ser feitos neste controle de tela. Como iremos fazer com que o usuário desenhe apenas quando a mão dele estiver fechada, devemos marcar como verdadeiro a propriedade `IsGripTarget`, com ela definida como verdadeiro o componente passa a identificar o estado da mão (aberta, fechada ou recém aberta).

No corpo do método devemos apenas criar o objeto do tipo `InteractionInfo` e marcar a propriedade citada acima, seu código deve ficar similar ao código a seguir.

```
public class CanvasInteracao : Canvas, IInteractionClient
{
    public InteractionInfo GetInteractionInfoAtLocation
     (int skeletonTrackingId, InteractionHandType handType,
      double x, double y)
    {
        InteractionInfo info = new InteractionInfo();
        info.IsGripTarget = true;

        return info;
    }
}
```

Agora podemos utilizar este componente em nossa aplicação da mesma forma que utilizamos um `Canvas`. Não iremos substituir o canvas que já está na aplicação, mas apenas inserir este nosso componente acima do `Canvas` já existente, pois o canvas já existente precisa limpar e redesenhar os componentes a cada quadro e este novo manterá o desenho enquanto o usuário estiver no modo pintura. Então basta

colocá-lo na mesma posição do `Canvas` que já existe. Além disso, iremos criar mais um `KinectToggleButton` para ativar a função de desenhar. Para que o usuário não crie desenhos por engano iremos habilitar esta funcionalidade apenas quando o botão mencionado estiver ativo, o arquivo XAML de sua janela completa deve ficar similar ao a seguir.

```
<Window x:Class="Interacao.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:k="http://schemas.microsoft.com/kinect/2013"
    xmlns:my="clr-namespace:Interacao.Auxiliar"
    Title="MainWindow" Height="800" Width="1000"
    WindowState="Maximized">
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="25"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="50"/>
        <RowDefinition Height="240*"/>
        <RowDefinition Height="180"/>
    </Grid.RowDefinitions>

    <Canvas Name="canvasKinect"
     Grid.Row="1" Grid.ColumnSpan="1"/>

    <my:CanvasInteracao x:Name="canvasDesenho"
     Grid.Row="1" Grid.ColumnSpan="1"/>

    <k:KinectRegion Name="kinectRegion"
     Grid.Row="2" Grid.ColumnSpan="2">
        <DockPanel>

            <k:KinectTileButton Content="Fechar"
             Height="130" Width="150" Foreground="White"
             DockPanel.Dock="Right" Click="btnFecharClick"/>

            <StackPanel  Orientation="Horizontal" >

                <my:KinectToggleButton x:Name="btnEscalaCinza"
```

```
                 Content="Escala Cinza" Height="130"
                 Foreground="White"/>

                <my:KinectToggleButton x:Name="btnEsqueletoUsuario"
                 Content="Esqueleto do Usuário" Height="130"
                 FontSize="20" Foreground="White"/>

                <my:KinectToggleButton x:Name="btnDesenhar"
                 Content="Desenhar" Height="130"
                 Foreground="White" Click="btnDesenharClick"/>

                <k:KinectTileButton Content="Desligar"
                 Height="130" Width="150"
                 Foreground="White" Click="btnVoltarClick"/>

            </StackPanel>
        </DockPanel>
    </k:KinectRegion>

    <k:KinectSensorChooserUI Name="seletorSensorUI"
     HorizontalAlignment="Center" VerticalAlignment="Top" />

    <Slider Name="slider" Width="20" Orientation="Vertical"
     Minimum="-27" Maximum="27" SmallChange="1" Value="0"
     Height="{Binding ElementName=imagemCamera, Path=ActualHeight}"
     Thumb.DragCompleted="slider_DragCompleted" Grid.Column="2"
     Grid.Row="1" Margin="3,0,2,0"/>

</Grid>
</Window>
```

Se você prestou atenção no código de nossa janela verá que o `KinectToggleButton btnDesenhar` possui um método associado ao evento de clique. Este método será utilizado apenas para limpar o desenho quando o usuário desativar a função desenho na tela, conforme o código a seguir.

```
private void btnDesenharClick(object sender, RoutedEventArgs e)
{
    if (!btnDesenhar.IsChecked)
        canvasDesenho.Children.Clear();
}
```

Também precisaremos adicionar ao namespace `Auxiliar` uma classe chamada `ConfiguracaoDesenho`. Como o próprio nome já sugere, esta classe terá a responsabilidade de configurar o desenho (forma, cor e tamanho). Em nossa aplicação, iremos utilizar dois configuradores, um para cada mão do usuário. Para definir cor e tamanho, temos objetos já preparados na linguagem, mas também devemos adicionar um enumerador (`enum`) para as formas. O enumerador será chamado de `FormaDesenho` e seu código deve ser semelhante ao código a seguir.

```
public enum FormaDesenho
{
    Retangulo,
    Elipse
}
```

Simples, não? Criamos isso para facilitar a identificação da forma que deve ser desenhada. Agora criaremos nossa classe de configuração, conforme o código.

```
public class ConfiguracaoDesenho
{
    public FormaDesenho Forma { get; set; }
    public SolidColorBrush Cor { get; set; }
    public int Tamanho { get; set; }
    public bool DesenhoAtivo {get; set;}
}
```

Esta classe apenas irá gerenciar a configuração do desenho, e não possuirá nenhum tipo de comportamento. Ainda precisamos fazer uma última coisa antes de implementarmos a interação com nosso novo `Canvas`: precisamos de um método que crie o desenho de acordo com as configurações da classe criada anteriormente.

Para a criação deste novo método vamos reaproveitar nosso código escrito na classe `EsqueletoUsuarioAuxiliar`, vários trechos já escritos no corpo de um método podem ser refatorados para permitir sua reutilização. Vamos analisar nosso código já existente no método `DesenharArticulacao`.

```
public void DesenharArticulacao
 (Joint articulacao, Canvas canvasParaDesenhar)
{
    int diametroArticulacao =
     articulacao.JointType == JointType.Head ? 50 : 10;
```

```
    int larguraDesenho = 4;
    Brush corDesenho = Brushes.Red;
    Ellipse objetoArticulacao =
     CriarComponenteVisualArticulacao
     (diametroArticulacao, larguraDesenho, corDesenho);

    ColorImagePoint posicaoArticulacao =
     ConverterCoordenadasArticulacao(articulacao,
     canvasParaDesenhar.ActualWidth, canvasParaDesenhar.ActualHeight);

    double deslocamentoHorizontal =
     posicaoArticulacao.X - objetoArticulacao.Width / 2;

    double deslocamentoVertical =
     (posicaoArticulacao.Y - objetoArticulacao.Height / 2);

    if ( deslocamentoVertical >= 0 &&
         deslocamentoVertical < canvasParaDesenhar.ActualHeight &&
         deslocamentoHorizontal >= 0 &&
         deslocamentoHorizontal < canvasParaDesenhar.ActualWidth )
    {
        Canvas.SetLeft
         (objetoArticulacao, deslocamentoHorizontal);

        Canvas.SetTop
         (objetoArticulacao, deslocamentoVertical);

        Canvas.SetZIndex
         (objetoArticulacao, 100);

        canvasParaDesenhar.Children.Add(objetoArticulacao);
    }

}
```

Analisando o código anterior, notamos que para desenhar uma articulação precisamos criar algumas configurações, depois disso criamos um componente visual (no caso uma elipse) de acordo com as configurações citadas anteriormente e por fim posicionamos o componente no `Canvas`.

A configuração do componente e seu posicionamento não estão diretamente relacionados com a operação de desenhar uma articulação. Vamos se-

parar estes passos em métodos menores e faremos chamadas para eles de dentro do método maior. A primeira coisa a ser feita é alterar o método `CriarComponenteVisualArticulacao` para que ele não crie mais um objeto do tipo `Ellipse` em seu corpo e passe a apenas configurar um componente visual informado por parâmetro, facilitando assim seu reuso, conforme o código abaixo.

```
private void ConfigurarComponenteVisualArticulacao
 (Shape forma, int diametroArticulacao,
  int larguraDesenho, Brush corDesenho)
{
    forma.Height = diametroArticulacao;
    forma.Width = diametroArticulacao;
    forma.StrokeThickness = larguraDesenho;
    forma.Stroke = corDesenho;
}
```

O objeto passado para o método exibido anteriormente é do tipo `Shape`. Desta forma, podemos configurar diversos tipos de formas e não apenas objetos circulares, como era feito anteriormente. Agora precisamos criar um novo método para executar os passos necessários para posicionar o elemento no `Canvas`. Chamaremos este método de `PosicionarDesenho`. Nele, será necessário receber por parâmetro a articulação que irá ser desenhada, o `Canvas` no qual o componente será desenhado e o componente propriamente dito, conforme o código abaixo.

```
private void PosicionarDesenho
 (Joint articulacao, Canvas canvasParaDesenhar, Shape objetoArticulacao)
{
    ColorImagePoint posicaoArticulacao =
     ConverterCoordenadasArticulacao(articulacao,
     canvasParaDesenhar.ActualWidth, canvasParaDesenhar.ActualHeight);

    double deslocamentoHorizontal =
     posicaoArticulacao.X - objetoArticulacao.Width / 2;

    double deslocamentoVertical =
     (posicaoArticulacao.Y - objetoArticulacao.Height / 2);

    if ( deslocamentoVertical >= 0 &&
         deslocamentoVertical < canvasParaDesenhar.ActualHeight &&
         deslocamentoHorizontal >= 0 &&
         deslocamentoHorizontal < canvasParaDesenhar.ActualWidth)
```

```
    {
        Canvas.SetLeft
         (objetoArticulacao, deslocamentoHorizontal);

        Canvas.SetTop
         (objetoArticulacao, deslocamentoVertical);

        Canvas.SetZIndex
         (objetoArticulacao, 100);

        canvasParaDesenhar.Children.Add(objetoArticulacao);
    }
}
```

E o método `DesenharArticulacao` deve ficar semelhante ao seguinte código.

```
public Ellipse DesenharArticulacao
 (Joint articulacao, Canvas canvasParaDesenhar)
{
    int diametroArticulacao =
     articulacao.JointType == JointType.Head ? 50 : 10;

    int larguraDesenho = 4;
    Brush corDesenho = Brushes.Red;

    Ellipse objetoArticulacao = new Ellipse();

    ConfigurarComponenteVisualArticulacao
     (objetoArticulacao, diametroArticulacao,
      larguraDesenho, corDesenho);

    PosicionarDesenho(articulacao, canvasParaDesenhar,
     objetoArticulacao);

    return objetoArticulacao;
}
```

O método acima passou a ter um corpo muito menor e suas responsabilidades foram separadas, o que facilita a criação de novos métodos para desenhar elementos em um canvas. Iremos reaproveitá-las agora para criar o método `InteracaoDesenhar`. Este método irá receber por parâmetro uma articulação,

um `Canvas` e um objeto de configuração de desenho — através deles podemos obter toda informação necessária para desenharmos nossas formas em no painel.

O corpo deste novo método ficará bastante similar ao método `DesenharArticulacao`, porém, diferente do citado, o objeto que será desenhado depende da configuração e poderá não ser do tipo `Ellipse`. Segue o código deste novo método.

```
public Shape InteracaoDesenhar
 (Joint articulacao, Canvas canvasParaDesenhar,
  ConfiguracaoDesenho configuracao)
{
    int larguraDesenho = 4;

    Shape objetoArticulacao;
    if (configuracao.Forma == FormaDesenho.Elipse)
        objetoArticulacao = new Ellipse();
    else
        objetoArticulacao = new Rectangle();

    ConfigurarComponenteVisualArticulacao(objetoArticulacao,
     configuracao.Tamanho, larguraDesenho, configuracao.Cor);

    objetoArticulacao.Fill = configuracao.Cor;

    PosicionarDesenho(articulacao, canvasParaDesenhar,
     objetoArticulacao);

    return objetoArticulacao;
}
```

Agora já temos tudo que precisamos para alterar o código de nossa janela, mãos à obra! Como já foi dito no início do capítulo, o fluxo de interação é um pouco diferente dos demais, então precisamos criar em nossa classe um objeto do tipo `InteractionStream` para gerenciá-lo. Além disso, iremos criar dois objetos configuradores, um para cada mão.

```
public partial class MainWindow : Window
{
    KinectSensor kinect;
    List<IRastreador> rastreadores;
    InteractionStream fluxoInteracao;
```

```
    ConfiguracaoDesenho configuracaoMaoDireita;
    ConfiguracaoDesenho configuracaoMaoEsquerda;

...
```

Seguindo os mesmos princípio dos outros objetos que gerenciamos em nossa janela, vamos criar um método para inicializar os configuradores, chamado `InicializarConfiguracoesDesenho`, conforme o código.

```
private void InicializarConfiguracoesDesenho()
{
    configuracaoMaoDireita = new ConfiguracaoDesenho();
    configuracaoMaoDireita.Cor = Brushes.Red;
    configuracaoMaoDireita.Forma = FormaDesenho.Elipse;
    configuracaoMaoDireita.Tamanho = 20;

    configuracaoMaoEsquerda = new ConfiguracaoDesenho();
    configuracaoMaoEsquerda.Cor = Brushes.Blue;
    configuracaoMaoEsquerda.Forma = FormaDesenho.Retangulo;
    configuracaoMaoEsquerda.Tamanho = 20;
}
```

Este é apenas um exemplo de configuração, esteja livre para colocar as cores, formas e tamanhos que desejar e não esqueça de inserir a chamada deste método no construtor da janela. Agora precisamos também inicializar o nosso fluxo de interação. Para que ele seja inicializado, é necessário ter um objeto que represente o sensor e um objeto que represente um `IInteractionClient`. Já temos os dois disponíveis em nossa janela, mas lembre-se que o sensor pode ser desconectado ou inserido a qualquer momento enquanto a aplicação estiver rodando, então, para garantir que tudo continue funcionando, precisaremos reinicializar o fluxo de interação sempre que um novo sensor for conectado, o que quer dizer que a chamada para sua inicialização deve estar no método `InicializarKinect`.

Como todos os outros fluxos, este também possui um evento para quando um novo quadro está pronto, que se chama `InteractionFrameReady`. Nele, podemos obter um objeto do tipo `InteractionFrame` e através dele obtermos as informações do usuário (`UserInfo`) que possui diversas informações, como por exemplo, o estado das mãos do usuário.

Para capturar estas informações do quadro de interação, executamos passos similares aos utilizados para capturamos o esqueleto do usuário através de seu quadro. Precisamos copiar toda informação para um array e logo em seguida fazer

uma validação neste array para encontrar o objeto referente ao usuário que está sendo rastreado. Após fazer isso, podemos extrair as informações através do array `HandPointers`. Este array possui somente duas posições e cada um de seus elementos possui informações a respeito de uma das mãos do usuário. Através da propriedade `HandEventType` de cada mão podemos verificar seu estado. Neste evento apenas iremos alterar a propriedade `DesenhoAtivo` dos configuradores de desenho, para que o desenho fique ativo somente quando o usuário fechar a mão, conforme o código.

```
private void InicializarFluxoInteracao()
{
    fluxoInteracao =
     new InteractionStream(kinect, canvasDesenho);

    fluxoInteracao.InteractionFrameReady +=
     fluxoInteracao_InteractionFrameReady;
}

private void fluxoInteracao_InteractionFrameReady
 (object sender, InteractionFrameReadyEventArgs e)
{
    using (InteractionFrame quadro = e.OpenInteractionFrame())
    {
        if (quadro == null) return;

        UserInfo[] informacoesUsuarios = new UserInfo[6];
        quadro.CopyInteractionDataTo(informacoesUsuarios);

        IEnumerable<UserInfo> usuariosRastreados =
         informacoesUsuarios.Where(
         info => info.SkeletonTrackingId != 0);

        if (usuariosRastreados.Count() > 0)
        {
            UserInfo usuarioPrincipal =
             usuariosRastreados.First();

            if ( usuarioPrincipal.HandPointers[0].HandEventType ==
                 InteractionHandEventType.Grip)
                configuracaoMaoDireita.DesenhoAtivo = true;
```

```
            else if ( usuarioPrincipal.HandPointers[0].HandEventType ==
                      InteractionHandEventType.GripRelease)
                configuracaoMaoEsquerda.DesenhoAtivo = false;

            if ( usuarioPrincipal.HandPointers[1].HandEventType ==
                 InteractionHandEventType.Grip)
                configuracaoMaoDireita.DesenhoAtivo = true;

            else if ( usuarioPrincipal.HandPointers[1].HandEventType ==
                      InteractionHandEventType.GripRelease)
                configuracaoMaoEsquerda.DesenhoAtivo = false;

        }
    }
}
```

Neste ponto encontramos a maior diferença entre o fluxo de interação e os outros, diferente dos fluxos padrões do sensor, o fluxo de interação não é processado automaticamente no sensor, por exemplo, o fluxo de profundidade é dependente do fluxo de esqueleto para poder obter o *player segmentation data* que já vimos anteriormente neste livro, mas para que essa dependência seja cumprida, basta que o fluxo de esqueleto esteja ligado. Esta forma automática de suprir dependências não ocorre no fluxo de interação — precisamos explicitar o processamento dos dados, tanto no quadro de esqueleto, quanto no quadro de profundidade.

Para fazermos com que estas informações sejam processadas, teremos de refatorar os métodos que utilizam os quadros de profundidade e esqueleto. Começaremos com a profundidade. Até o momento nossa função para profundidade é utilizada apenas quando a função de escala cinza está ativa, precisamos alterar este comportamento para que o método que utiliza o quadro de profundidade sempre seja chamado, permitindo que se processe a informação necessária para o fluxo de interação.

Vamos primeiro renomear o método de `ReconhecerProfundidade` para `FuncoesProfundidade` assim como fizemos com o esqueleto. Depois de renomearmos, criaremos um novo método chamado `ReconhecerProfundidade`, que deve ser chamado pelo `FuncoesProfundidade` e executar apenas a parte em que a regra para reconhecer profundidade é aplicada, tornando o método idêntico ao código a seguir.

```
private void FuncoesProfundidade
 (DepthImageFrame quadro, byte[] bytesImagem, int distanciaMaxima)
```

```
{
    if (quadro == null || bytesImagem == null) return;

    using (quadro)
    {
        DepthImagePixel[] imagemProfundidade =
         new DepthImagePixel[quadro.PixelDataLength];

        quadro.CopyDepthImagePixelDataTo(imagemProfundidade);

        if (btnEscalaCinza.IsChecked)
            ReconhecerProfundidade
             (bytesImagem, distanciaMaxima, imagemProfundidade);
    }
}

private void ReconhecerProfundidade
 (byte[] bytesImagem, int distanciaMaxima,
  DepthImagePixel[] imagemProfundidade)
{
    DepthImagePoint[] pontosImagemProfundidade =
        new DepthImagePoint[640 * 480];

        kinect.CoordinateMapper
              .MapColorFrameToDepthFrame(kinect.ColorStream.Format,
              kinect.DepthStream.Format, imagemProfundidade,
              pontosImagemProfundidade);

        for (int i = 0; i < pontosImagemProfundidade.Length; i++)
        {
            var point = pontosImagemProfundidade[i];
            if ( point.Depth < distanciaMaxima &&
                 KinectSensor.IsKnownPoint(point) )
            {
                var pixelDataIndex = i * 4;

                byte maiorValorCor =
                    Math.Max(bytesImagem[pixelDataIndex],
                            Math.Max(bytesImagem[pixelDataIndex + 1],
                            bytesImagem[pixelDataIndex + 2]));
```

```
                bytesImagem[pixelDataIndex] = maiorValorCor;
                bytesImagem[pixelDataIndex + 1] = maiorValorCor;
                bytesImagem[pixelDataIndex + 2] = maiorValorCor;
        }
    }
}
```

Você não deve se esquecer de remover a condição da chamada deste método no evento `kinect_AllFramesReady`. Agora este método deve ser chamado sempre que houver um novo quadro disponível. Quando o usuário ativar a função de desenho, tanto o filtro de escala cinza quanto a exibição do esqueleto serão cancelados, então estes métodos não devem ser chamados caso a função de desenho esteja ativa. Vamos cobrir esta restrição e inserir o trecho que faz o processamento dos dados de interação, ambos serão feitos no método `FuncoesProfundidade` dentro do bloco `using`.

```
...
using (quadro)
    {
        DepthImagePixel[] imagemProfundidade =
         new DepthImagePixel[quadro.PixelDataLength];

        quadro.CopyDepthImagePixelDataTo(imagemProfundidade);

        if (btnDesenhar.IsChecked)
            fluxoInteracao.ProcessDepth
             (imagemProfundidade, quadro.Timestamp);

        else if (btnEscalaCinza.IsChecked)
            ReconhecerProfundidade
             (bytesImagem, distanciaMaxima, imagemProfundidade);
    }
...
```

Com isso nós forçamos o processamento dos dados de interação. Agora precisamos fazer uma operação similar no método `FuncoesEsqueletoUsuario`, mas neste último caso, também é necessário invocar os métodos para desenhar as formas, caso os desenhos estejam ativos, conforme o código a seguir.

```
private void FuncoesEsqueletoUsuario(SkeletonFrame quadro)
{
```

```
    if (quadro == null) return;

    using (quadro)
    {
        Skeleton esqueletoUsuario =
         quadro.ObterEsqueletoUsuario();

        if (btnDesenhar.IsChecked)
        {
            Skeleton[] esqueletos = new Skeleton[6];
            quadro.CopySkeletonDataTo(esqueletos);

            fluxoInteracao.ProcessSkeleton(esqueletos,
             kinect.AccelerometerGetCurrentReading(),
             quadro.Timestamp);

            EsqueletoUsuarioAuxiliar esqueletoAuxiliar =
             new EsqueletoUsuarioAuxiliar(kinect);

            if (configuracaoMaoDireita.DesenhoAtivo)
                esqueletoAuxiliar.InteracaoDesenhar
                 (esqueletoUsuario.Joints[JointType.HandRight],
                 canvasDesenho, configuracaoMaoDireita);

            if (configuracaoMaoEsquerda.DesenhoAtivo)
                esqueletoAuxiliar.InteracaoDesenhar
                 (esqueletoUsuario.Joints[JointType.HandLeft],
                 canvasDesenho, configuracaoMaoEsquerda);
        }
        else
        {
            foreach (IRastreador rastreador in rastreadores)
                rastreador.Rastrear(esqueletoUsuario);

            if (btnEsqueletoUsuario.IsChecked)
                quadro.DesenharEsqueletoUsuario(kinect, canvasKinect);
        }
    }
}
```

Agora já podemos executar nosso aplicativo e ver os resultados, crie e teste di-

versas configurações de desenhos diferentes, você irá notar que como quase tudo no sensor, há uma certa imprecisão, mas é possível ter ótimos resultados desenhando com o Kinect!
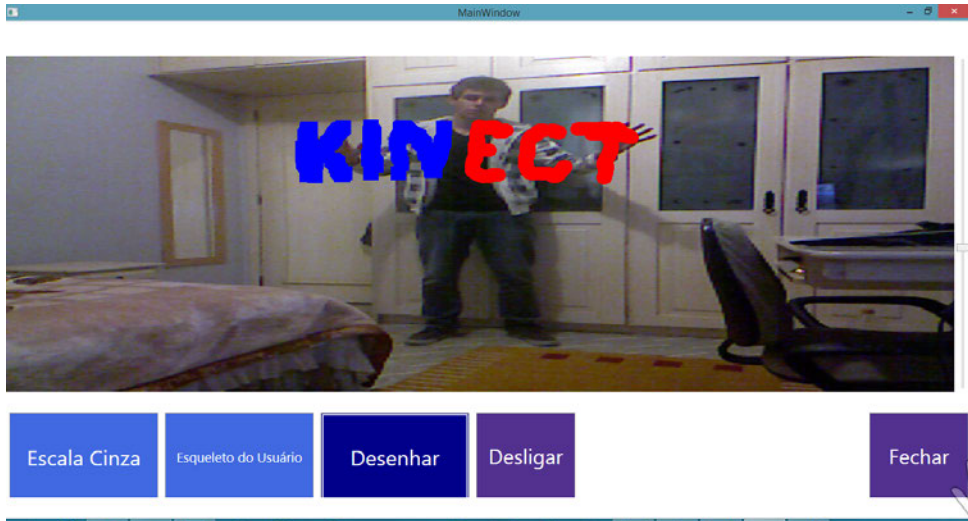
Figura 8.3: Desenhando no Canvas utilizando Interactions

Capítulo 9

Utilizando os Microfones do Sensor

Além de toda a parte de movimentos, o sensor Kinect também conta com quatro microfones e com uma capacidade grande para identificar os sons do ambiente. Essa identificação pode ser separada em duas partes: identificação da **direção do som** e a de **palavras e frases**.

Durante o processo de captar sons do ambiente através do Kinect, é possível fazer o cancelamento de 20 decibéis de altura. Isso ajuda a suprimir o som ambiente e melhorar a fidelidade das informações captadas. Além disso, é possível configurar programaticamente o sensor para captar os sons de uma determinada direção. Por padrão, o sensor não dá prioridade para nenhuma direção e altera a prioridade para a fonte de áudio com o volume mais alto.

A utilização de comandos de voz com o sensor Kinect permite que a aplicação compreenda algumas palavras e frases e execute ações baseadas nisso. Esta forma de interação pode ser muito natural ao usuário, mas o design deste tipo de interface deve ser bem projetado para retornar um resultado satisfatório. Um erro bastante comum é a utilização de palavras com a fonética similar — sempre que você encontrar a

necessidade de palavras similares busque por sinônimos com fonéticas diferentes, para evitar erros na interpretação de comandos.

Ao longo do sensor, os microfones atingem um raio total de 100 graus pela frente e a direção do som que está vindo da fonte do áudio é identificada dentro desta angulação. Desta forma, podemos fazer com que a aplicação perceba a direção de sons, conforme ilustra a figura 9.1.
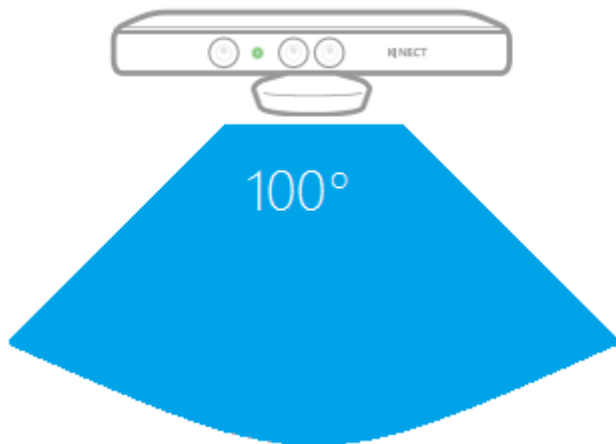
Figura 9.1: Raio de Detecção de Som do Kinect

9.1 Inicializando a Fonte de Áudio e Detectando a Direção do Som

O primeiro processo que iremos fazer é a inicialização da fonte de áudio do sensor. A fonte de áudio é utilizada para ligar os microfones e para obter o fluxo de entrada de som. Existem diversas configurações que podemos fazer na fonte de áudio, como por exemplo, a já mencionada supressão de ruídos através do cancelamento de 20 decibéis.

Para capturar o áudio, o Kinect recebe sons do ambiente por diversas direções. Entretanto, há a propriedade `BeamAngle` que corresponde a uma faixa de 10 graus em que o Kinect está apto a captar sons melhores. As ondas de som que são propagadas dentro desta faixa são separadas das ondas que se propagam fora dela. Você pode utilizar esta propriedade para atribuir manualmente uma direção que seja in-

teressante para sua aplicação prestar mais atenção e isolar o áudio, ou você pode utilizar as formas automáticas.

Nas formas automáticas (`Automatic` e `Adaptive`), esta faixa de foco é atualizada constantemente e alterada para a direção onde a fonte de áudio é mais alta. Por padrão, a forma `Automatic` é selecionada. Ela deve ser utilizada para ambientes com pouco barulho e ruído. Já a forma `Adaptive`, apesar de também ser automatizada, é mais indicada para locais barulhentos.

Além do `BeamAngle`, existe a propriedade `SoundSourceAngle` que muitas vezes tem suas funcionalidades confundidas. Esta propriedade é atualizada com uma frequência bem maior, pois ela indica a direção de onde um novo ruído está vindo, ou seja, sempre que há um ruído perceptível pelo Kinect esta propriedade é alterada. Em casos em que o desenvolvedor opta por utilizar o `BeamAngle` manual, é comum utilizar o `SoundSourceAngle` para ajustar sua posição.

A primeira parte de nossa implementação será inicializar a fonte de áudio do sensor, começar a captar a direção do áudio e criar uma forma visual para mostrar isso ao usuário, tanto o `BeamAngle` quanto o `SoundSourceAngle`. Faremos isso utilizando uma barra similar à barra de progresso de nossa pose, e criaremos também dois indicadores, um para cada propriedade.

Vamos ao arquivo XAML de nossa janela principal e iremos alterar a altura da ultima linha do `Grid` de 180 para 200. A barra que indicará a direção do som ficará acima dos botões previamente criados. Depois de fazer isso, iremos circundar o componente `KinectRegion` com um novo `Grid`. Este painel deve possui duas linhas, uma com 20 pixels de altura e outra com todo o restante (*). O `KinectRegion` deve ocupar a segunda linha com todo o espaço restante, enquanto que na primeira linha criaremos um canvas para mostrar a barra e os indicadores mencionados anteriormente.

O trecho no qual antes havia apenas o `KinectRegion` deve ficar similar ao trecho de código a seguir.

```
<Grid Grid.Row="2" Grid.ColumnSpan="1">

    <Grid.RowDefinitions>
        <RowDefinition Height="20"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <Canvas Name="barraDirecaoAudio" Background="BlueViolet">
```

```
        <Rectangle Name="ponteiroBeamAngle"
         Width="7" Height="20" Fill="DarkGray"/>

        <Rectangle Name="ponteiroSoundSourceAngle"
        Width="3" Height="20" Fill="White"/>

    </Canvas>

    <k:KinectRegion Name="kinectRegion" Grid.Row="2" Grid.ColumnSpan="2">
    <DockPanel>

        <k:KinectTileButton Content="Fechar"
         Height="130" Width="150" Foreground="White"
         DockPanel.Dock="Right" Click="btnFecharClick"/>

        <StackPanel  Orientation="Horizontal" >

            <my:KinectToggleButton x:Name="btnEscalaCinza"
             Content="Escala Cinza" Height="130"
             Foreground="White"/>

            <my:KinectToggleButton x:Name="btnEsqueletoUsuario"
             Content="Esqueleto do Usuário" Height="130"
             FontSize="20" Foreground="White"/>

            <my:KinectToggleButton x:Name="btnDesenhar"
             Content="Desenhar" Height="130"
             Foreground="White" Click="btnDesenharClick"/>

            <k:KinectTileButton Content="Desligar"
             Height="130" Width="150"
             Foreground="White" Click="btnVoltarClick"/>

        </StackPanel>
    </DockPanel>
</k:KinectRegion>
</Grid>
```

Com o código anterior, já temos a parte visual de nossa implementação. Agora o que iremos fazer é inicializar a fonte de áudio do sensor e posicionar os ponteiros

sempre que as propriedades forem alteradas. Tudo que tange a fonte de áudio do sensor pode ser acessado através da propriedade `AudioSource` de um objeto do tipo `KinectSensor`. No caso de nossa aplicação, criaremos um método chamado `InicializarFonteAudio`. Nele, iremos ativar a supressão e cancelamento de eco, interpretar os eventos de alteração no valor das propriedades e inicializar a fonte de áudio propriamente dita. Seu código deve ficar similar ao código a seguir.

```
private void InicializarFonteAudio()
{
    kinect.AudioSource.EchoCancellationMode =
     EchoCancellationMode.CancellationAndSuppression;

    kinect.AudioSource.BeamAngleChanged
     += AudioSource_BeamAngleChanged;

    kinect.AudioSource.SoundSourceAngleChanged
     += AudioSource_SoundSourceAngleChanged;

    kinect.AudioSource.Start();
}

private void AudioSource_SoundSourceAngleChanged
 (object sender, SoundSourceAngleChangedEventArgs e)
{

}

private void AudioSource_BeamAngleChanged
 (object sender, BeamAngleChangedEventArgs e)
{

}
```

Você também deve lembrar-se de inserir uma chamada ao método `InicializarFonteAudio` no método `InicializarKinect`. Agora iremos implementar a lógica para posicionar os ponteiros no local correto de acordo com o valor da propriedade respectiva (`BeamAngle` ou `SoundSourceAngle`).

O cálculo para que o ponteiro fique em sua posição é bastante simples, apenas iremos fazer a conhecida regra de três para escalar o valor do ângulo que varia de -50 até 50 graus, para o tamanho da barra que exibe os ponteiros, que varia de 0 até

a largura atual da barra. A regra de três geralmente inicia com uma escala de zero até determinado número, em nosso caso não será diferente — apesar da escala de ângulos variar de -50 até 50, podemos assumir que ela varia de 0 até 100, ou seja, manteremos a proporção no momento da regra de três.

Após a regra de três, é necessário somar ao resultado o valor que corresponde à metade da largura. Essa soma se faz necessária para reajustar a escala de ângulos. Por fim, diminuiremos o deslocamento correspondente à metade da largura do ponteiro, para que o mesmo fique no centro da posição.

Além disso, deve ser feito um reajuste para as bordas, ou seja, o componente não deve estar em uma posição menor que zero e nem alta o suficiente para que ele ultrapasse a borda da direita. Iremos encapsular estas regras em um método chamado `AjustarPonteiroPorAngulo`, que receberá por parâmetro um objeto do tipo `double`, correspondente ao valor do ângulo e um objeto do tipo `Rectangle`, correspondente ao ponteiro da propriedade. Seguindo estes passos, seu código deve ficar similar ao código a seguir.

```
private void AjustarPonteiroPorAngulo(double angulo, Rectangle ponteiro)
{
    double calculoPosicao =
     (barraDirecaoAudio.ActualWidth * angulo / 100) +
     (barraDirecaoAudio.ActualWidth / 2) -
     ponteiro.ActualWidth / 2;

    double deslocamento =
     Math.Min(calculoPosicao,
            barraDirecaoAudio.ActualWidth - ponteiro.ActualWidth);

    deslocamento = Math.Max(calculoPosicao, 0);

    Canvas.SetLeft(ponteiro, deslocamento);
}
```

Em cada evento que interpretamos anteriormente iremos fazer uma chamada a este novo método, conforme o código.

```
private void AudioSource_SoundSourceAngleChanged
 (object sender, SoundSourceAngleChangedEventArgs e)
    {
        AjustarPonteiroPorAngulo(e.Angle, ponteiroSoundSourceAngle);
    }
```

```
private void AudioSource_BeamAngleChanged
 (object sender, BeamAngleChangedEventArgs e)
{
    AjustarPonteiroPorAngulo(e.Angle, ponteiroBeamAngle);
}
```

Você já pode testar sua aplicação! Os resultados devem ser semelhantes ao ilustrado pela figura 9.2
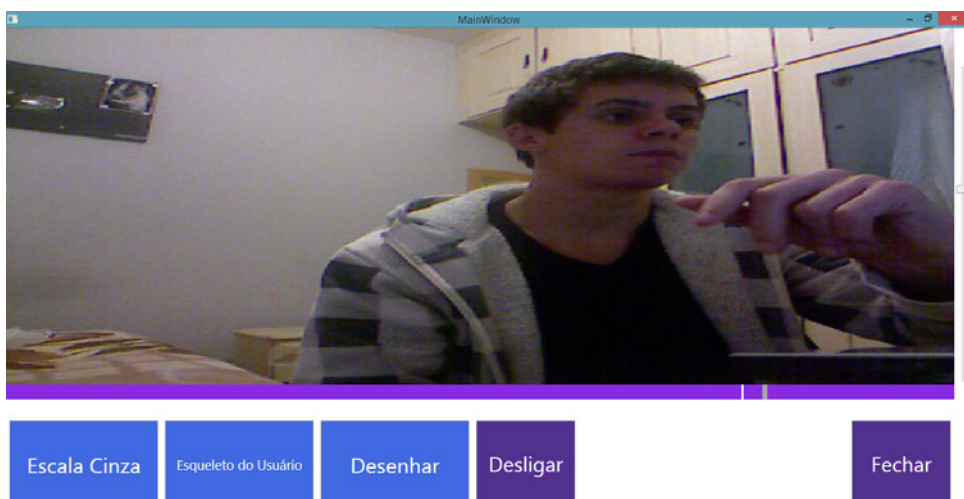
Figura 9.2: Detectando a direção do áudio através do Kinect

9.2 Detectando Comandos de voz

Agora que já aprendemos a captar a direção do áudio, vamos para a segunda parte da fonte de audio do Kinect. Vamos aprender a fazer com que nossa aplicação utilize o sensor para reconhecer comandos de voz e tomar ações baseadas nos comandos reconhecidos.

O reconhecimento de comandos de voz através do Kinect em uma aplicação .NET é feita através do engenho de reconhecimento de voz. É importante saber que este engenho funciona independente do sensor, por isso é necessário o download da DLL *Microsoft.Speech*. Há um pacote de extensão de idiomas para o reconhecimento de voz, mas infelizmente ainda não há opções para fazer o reconhecimento

de comandos em português, então utilizaremos o idioma padrão (inglês dos Estados Unidos).

É necessário entender que o sensor não irá compreender o áudio de forma livre e identificar tudo o que o usuário fala. Ao iniciarmos o engenho, iremos definir uma série de escolhas de comandos que nossa aplicação pode compreender e absolutamente tudo que o usuário falar o engenho tentará encaixar na melhor escolha entre as que existem. O resultado de uma interpretação possui o comando que foi escolhido e a confiança que a aplicação tem de que foi aquilo que o usuário disse.

O primeiro passo para detectar comandos de voz na aplicação é inicializar o engenho de reconhecimento de voz mencionado no parágrafo anterior. Para inicializá-lo, é necessário encontrar e reconhecer o idioma que será utilizado, carregar os comandos que serão intrepretados pela aplicação (o conjunto de comandos é chamado de gramática), inserir o fluxo do áudio do kinect no engenho e ativar de fato o reconhecimento.

Vamos implementar os passos descritos anteriormente em nossa DLL `AuxiliarKinect`, na classe `InicializadorKinect`. Da mesma forma que criamos uma propriedade para inserir o método de inicialização do sensor, faremos uma propriedade para inserir o método que gera a gramática do engenho. Depois disso, criaremos outra propriedade para armazenar o engenho propriamente dito e um método `InicializarReconhecimentoVoz`.

A propriedade `EngenhoReconhecimentoVoz` deve ser do tipo `SpeechRecognitionEngine`. A propriedade `MetodoInicializadorKinect` é do tipo `Action`, isso porque ela se refere a um método que não possui retorno. Caso o método possua um retorno, que é o caso da propriedade que será criada, é necessário utilizar o delegate `Func`. As propriedades devem ficar como o código a seguir.

```
public SpeechRecognitionEngine EngenhoReconhecimentoVoz
{ get; private set; }

public Func<Grammar> MetodoGerarGramatica
{ get; set; }
```

Agora já podemos vincular um método para gerar a gramática dos comandos em nosso inicializador, mas ainda precisamos criar o método que inicializa o reconhecimento de comandos de voz. Este método se chamará `InicializarReconhecimentoVoz` e deve receber um objeto do tipo `Stream`

por parâmetro, que conterá o fluxo de áudio do Kinect. Além disso, dentro do método faremos os passos descritos anteriormente.

```
public void InicializarReconhecimentoVoz(Stream fluxoAudio)
{
    Func<RecognizerInfo, bool> encontrarIdioma = reconhecedor =>
    {
        string value;
        reconhecedor.AdditionalInfo.TryGetValue("Kinect", out value);
        return "True".Equals(value, StringComparison
                     .InvariantCultureIgnoreCase) &&
                "en-US".Equals(reconhecedor.Culture.Name,
                 StringComparison.InvariantCultureIgnoreCase);
    };

    RecognizerInfo recognizerInfo =
    SpeechRecognitionEngine.InstalledRecognizers()
                           .Where(encontrarIdioma)
                           .FirstOrDefault();

    EngenhoReconhecimentoVoz =
     new SpeechRecognitionEngine(recognizerInfo.Id);

    EngenhoReconhecimentoVoz.LoadGrammar(MetodoGerarGramatica());

    EngenhoReconhecimentoVoz.SetInputToAudioStream
     (fluxoAudio,
      new SpeechAudioFormatInfo
      (EncodingFormat.Pcm, 16000, 16, 1, 32000, 2, null));

    EngenhoReconhecimentoVoz.RecognizeAsync(RecognizeMode.Multiple);
}
```

Como você pode ver, foi criada um função para buscar o idioma entre os idiomas instalados no engenho de reconhecimento de voz, e após isso o engenho foi inicializado passando as informações de reconhecimento. Também utilizamos a propriedade criada anteriormente para carregar a gramática e em seguida inserimos o fluxo de áudio do Kinect no engenho com as devidas configurações.

Os parâmetros informados no objeto `SpeechAudioFormatInfo` passado por parâmetro no método `SetInputToAudioStream` indicam a configuração do sensor Kinect para codificação de áudio, quantidade de exemplos por segundo, quan-

tidade de bits, por exemplo, quantidade de canais de áudio, média de bytes por segundo, alinhamento dos blocos de áudio e uma matriz de bytes contendo o formato específico dos dados, respectivamente.

E por fim, iniciamos o reconhecimento assíncrono de comandos de voz. Agora nossa DLL `AuxiliarKinect` já está pronta para ser utilizada em nossa janela. Voltamos novamente ao nosso projeto da aplicação e iremos fazer modificações nos métodos `InicializarSeletor` e `InicializarFonteAudio`. Além disso, devemos tornar o seletor do sensor um atributo da janela, para que ele possa ser acessado de qualquer método de nossa classe `MainWindow`.

O método `InicializarSeletor` deve ser refatorado para que ele atribua também a propriedade `MetodoGerarGramatica` para um método da janela. Por enquanto, deixaremos este método em branco, conforme o código a seguir.

```
private void InicializarSeletor()
{
    inicializador = new InicializadorKinect();
    inicializador.MetodoInicializadorKinect = InicializarKinect;
    inicializador.MetodoGerarGramatica = GerarGramatica;
    seletorSensorUI.KinectSensorChooser = inicializador.SeletorKinect;
}

private Grammar GerarGramatica()
{

}
```

O método `InicializarFonteAudio` também deve ser refatorado. Nós utilizamos o método `Start` da propriedade `AudioSource`, mas não precisávamos obter seu retorno. A partir de agora, iremos obter o fluxo de áudio que é retornado deste método para podermos inicializar no engenho de reconhecimento de voz. Após o inicializarmos, devemos interpretar o evento de comando reconhecido, conforme o código a seguir.

```
private void InicializarFonteAudio()
{
    kinect.AudioSource.EchoCancellationMode =
     EchoCancellationMode.CancellationAndSuppression;

    kinect.AudioSource.BeamAngleChanged
     += AudioSource_BeamAngleChanged;
```

```
    kinect.AudioSource.SoundSourceAngleChanged
     += AudioSource_SoundSourceAngleChanged;

    Stream fluxoAudio = kinect.AudioSource.Start();

    inicializador.InicializarReconhecimentoVoz(fluxoAudio);
    inicializador.EngenhoReconhecimentoVoz
                 .SpeechRecognized +=
                 KinectSpeechRecognitionEngine_SpeechRecognized;
}

private void KinectSpeechRecognitionEngine_SpeechRecognized
 (object sender, SpeechRecognizedEventArgs e)
{

}
```

No exemplo deste livro iremos tratar apenas comandos identificados, mas caso você queira, há outros tipos de eventos disponíveis, como por exemplo, toda vez que um comando é hipotetizado, ou rejeitado.

Antes de implementarmos o método de reconhecimento de comandos é necessário que os tenhamos definido e, como visto anteriormente, nosso método para gerar a gramática ainda está em branco e chegou a hora de o implementarmos. Existem dois comandos que quase sempre devem ser criados em uma gramática, que são utilizados para ativar e desativar a interpretação dos outros comandos.

Imagine o seguinte cenário: você está com sua aplicação aberta e está tendo uma conversa com alguém. Seria frustrante se sua aplicação executasse uma ação baseada em uma frase que você disse em sua conversa. Para evitar este problema geralmente há uma palavra chave — no caso do Kinect no console Xbox 360, a palavra chave para ativação de comandos de voz é "Xbox". Em nossa aplicação as palavras chaves serão: **Kinect** e **Cancel**, para ativar e desativar a interpretação de comandos.

No capítulo anterior, criamos uma maneira do usuário desenhar formas e cores usando o fluxo de interação, mas infelizmente, uma vez definida a configuração de desenho, ela permanece por toda a aplicação. Iremos utilizar comandos de voz para permitir a alteração das cores e formas de ambas as mãos do usuário. Para que o usuário altere uma cor ou uma forma, ele deve dizer uma sentença completa, indicando qual a mão que terá a configuração alterada, o que será alterado (cor ou forma)

e qual o valor que a propriedade assumirá. As sentenças ficarão no seguinte formato: “Change (left/right) hand (color/shape) to (valor)”.

Iremos criar dois métodos para retornarmos os valores possíveis de todas as cores e de todas as formas em uma lista de `string`. Neste exemplo iremos nos limitar a um pequeno grupo de opções. Você pode aumentar o tamanho da lista se desejar, mas a estrutura de seus métodos deve ficar semelhante ao código a seguir.

```
private List<string> GerarListaCores()
{
    List<string> cores = new List<string>();
    cores.Add("red");
    cores.Add("blue");
    cores.Add("yellow");
    cores.Add("green");
    cores.Add("pink");

    return cores;
}

private List<string> GerarListaFormas()
{
    List<string> formas = new List<string>();
    formas.Add("rectangle");
    formas.Add("ellipse");

    return formas;
}
```

Com isso, já temos os possíveis valores e a estrutura de nossos comandos, agora já é hora de implementarmos o método `GerarGramatica`. Para gerarmos uma gramática é necessário utilizar a classe `Choices`, que permite adicionarmos as escolhas de comandos que a aplicação irá compreender. Estes comandos podem ser adicionados em forma de strings. Após popularmos todas as escolhas que nossa aplicação possui, é necessário criar um objeto do tipo `GrammarBuilder` para construir a gramática utilizando os comandos e, por fim, criar um objeto do tipo `Grammar`, que é a gramática propriamente dita, conforme o código a seguir.

```
private Grammar GerarGramatica()
{
    Choices comandos = new Choices();
```

```
List<string> listaFormas = GerarListaFormas();
List<string> listaCores = GerarListaCores();
comandos.Add("Kinect");
comandos.Add("Cancel");

for (int indice = 0; indice < 2; indice++)
{
    StringBuilder sentencaBase = new StringBuilder();
    string direcaoMao = indice == 0 ? "right" : "left";

    sentencaBase.Append("Change ");
    sentencaBase.Append(direcaoMao);
    sentencaBase.Append(" hand ");

    string inicial = sentencaBase.ToString();

    for ( int indiceLista = 0;
          indiceLista < listaCores.Count;
          indiceLista++)
    {
        StringBuilder sentencaFinal =
         new StringBuilder(inicial);

        sentencaFinal.Append("color to ");
        sentencaFinal.Append(listaCores[indiceLista]);

        comandos.Add(sentencaFinal.ToString());
    }

    for ( int indiceLista = 0;
          indiceLista < listaFormas.Count;
          indiceLista++)
    {
        StringBuilder sentencaFinal =
         new StringBuilder(inicial);

        sentencaFinal.Append("shape to ");
        sentencaFinal.Append(listaFormas[indiceLista]);

        comandos.Add(sentencaFinal.ToString());
    }
```

```
    }

    GrammarBuilder construtor =
     new GrammarBuilder(comandos);

    construtor.Culture =
     inicializador.EngenhoReconhecimentoVoz.RecognizerInfo.Culture;

    Grammar gramatica = new Grammar(construtor);

    return gramatica;
}
```

Com o código anterior, nossa gramática já está pronta e nossa aplicação já está reconhecendo os comandos, porém ainda não tomamos nenhuma ação, independente do comando que é reconhecido. Agora iremos implementar o método associado ao evento de reconhecimento de comando de voz que criamos anteriormente.

Como já foi dito antes, toda vez que um comando de voz é reconhecido, o resultado enviado para nosso evento possui um nível de confiança. Este nível de confiança deve ser levado em consideração para evitar que ações sejam tomadas indesejadamente. O recomendado pela Microsoft é que os comandos reconhecidos devem ter um nível de confiança de 80% ou mais, mas como não estamos utilizando nosso idioma nativo, e o sotaque também é levado em consideração, iremos aceitar qualquer comando com nível de confiança de 50% ou mais.

Então, no método que é disparado quando a aplicação reconhece um comando de voz, é necessário validar o nível de confiança. Para fazer isso, utilize a propriedade `Confidence`, que varia de 0 até 1, sendo que 1 é equivalente a 100% de confiança. Após validar a confiança, é necessário verificar se o comando reconhecido é um comando chave, caso seja iremos notificar o usuário que nossa aplicação passou a escutar ou a não escutar os comandos de voz. Para fazer isso, utilizaremos a propriedade `IsListening` do componente visual `seletorSensorUI`. Caso não seja um comando chave, iremos passar o resultado adiante para um método que chamaremos de `InterpretarComandoConfiguracaoCor`, conforme o código a seguir.

```
private void KinectSpeechRecognitionEngine_SpeechRecognized
 (object sender, SpeechRecognizedEventArgs e)
{
```

```
    RecognitionResult resultado = e.Result;
    if (resultado.Confidence > 0.5)
    {
        if (resultado.Text == "Kinect")
            seletorSensorUI.IsListening = true;
        else if (resultado.Text == "Cancel")
            seletorSensorUI.IsListening = false;
        else if (seletorSensorUI.IsListening)
            InterpretarComandoConfiguracaoCor(resultado);
    }
}

private void InterpretarComandoConfiguracaoCor
 (RecognitionResult resultado)
{

}
```

Se você executar sua aplicação, já poderá notar que ao dizer o comando de voz **Kinect** ela passa a notificar o usuário que o sensor está pronto para escutar comandos de voz, conforme a figura 9.3.
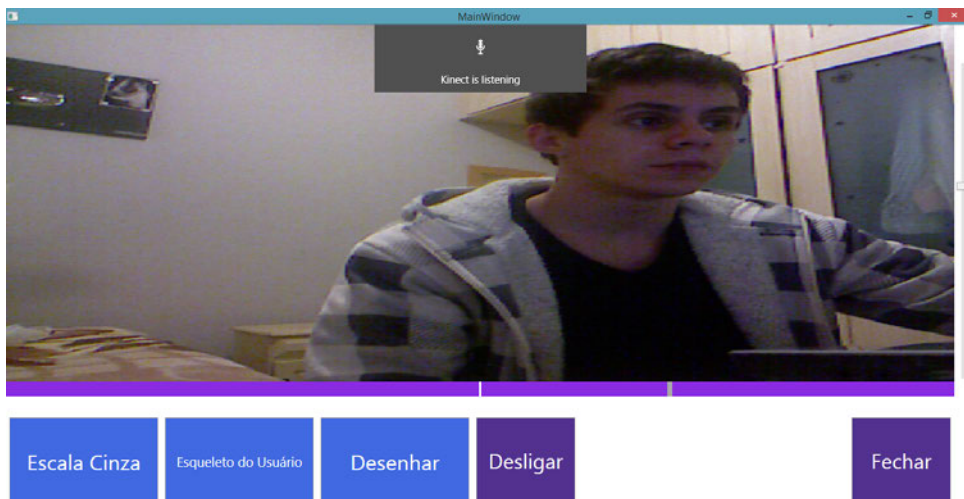
Figura 9.3: Kinect pronto para receber comandos de voz

Agora iremos implementar o método que interpreta os comandos de configura-

ção. Para fazer isso será bastante simples: primeiro, iremos verificar se o comando possui a subsentença "left hand" ou "right hand", com isso já descobrimos qual configuração devemos alterar. Depois, buscamos pela palavra "color" ou "shape" para descobrirmos qual a propriedade que deverá ser alterada. E por fim, buscamos a ultima palavra da sentença e aplicamos o seu valor para a respectiva propriedade, conforme o código a seguir.

```
private void InterpretarComandoConfiguracaoCor
 (RecognitionResult resultado)
{
    ConfiguracaoDesenho configuracao;

    if (resultado.Text.Contains("left hand"))
        configuracao = configuracaoMaoEsquerda;
    else
        configuracao = configuracaoMaoDireita;

    if (resultado.Text.Contains("color"))
    {
        string cor =
         resultado.Text.Substring(resultado.Text.LastIndexOf(" ") + 1);

        SolidColorBrush novaCor = Brushes.Red;
        switch (cor)
        {
            case "red":
                novaCor = Brushes.Red;
                break;
            case "blue":
                novaCor = Brushes.Blue;
                break;
            case "yellow":
                novaCor = Brushes.Yellow;
                break;
            case "green":
                novaCor = Brushes.Green;
                break;
            case "pink":
                novaCor = Brushes.Pink;
                break;
        }
```

```
        configuracao.Cor = novaCor;
    }
    else if (resultado.Text.Contains("shape"))
    {
        string forma =
         resultado.Text.Substring(resultado.Text.LastIndexOf(" ") + 1);

        FormaDesenho novaForma = FormaDesenho.Retangulo;
        switch (forma)
        {
            case "rectangle":
                novaForma = FormaDesenho.Retangulo;
                break;
            case "ellipse":
                novaForma = FormaDesenho.Elipse;
                break;
        }

        configuracao.Forma = novaForma;
    }
}
```

Agora nossa aplicação já pode receber alterações de configuração do desenho das mãos em tempo de execução via comandos de voz! Você pode criar diferentes resultados, como por exemplo, o resultado ilustrado na figura 9.4.
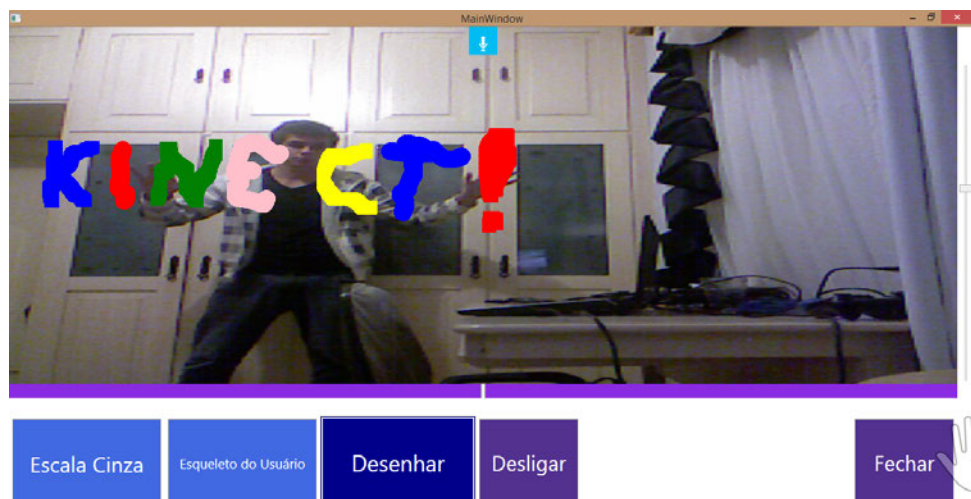
Figura 9.4: Pintura Final

9.3 Conclusão

Terminando a implementação anterior, nós finalizamos a aplicação deste livro. Durante a leitura passamos pelas funcionalidades mais importantes do sensor de movimentos Kinect sempre focando na didática. Os padrões arquiteturais de aplicação foram totalmente ignorados, com o objetivo de manter você concentrado apenas no sumo do conteúdo. Um de meus objetivos ao escrever este livro é disparar um gatilho para sua busca por conhecimento. Este material possui apenas uma parcela de todo o conteúdo do mundo NUI, até mesmo as implementações referentes ao sensor Kinect não foram abordadas em 100%.

O desenvolvimento com o Kinect é uma tarefa bastante complexa e divertida: acredito ser uma ótima forma de unirmos matemática, criatividade e programação, sendo que a maior limitação é sua imaginação. Busque também por códigos não mencionados aqui, como por exemplo, rastreamento de face, compreensão de semântica em comandos de voz e geração de gramática por arquivo XML. Toda a área de NUI ainda é muito nova e inexplorada e é certo que teremos diversas novidades tanto em software como em hardware para buscar a melhor usabilidade natural das aplicações.

Como autor eu busquei compartilhar de forma didática, simples e enxuta o máximo de informações e ideias com o intuito de auxiliar os mais diversos programa-

dores que buscam ingressar no mundo da NUI através do sensor Microsoft Kinect.